Influence-Based Voronoi Diagrams of Clusters[☆]

Ziyun Huang

Department of Computer Science and Software Engineering
Penn State Erie, the Behrend College
4701 College Drive, Erie, PA 16563, 14260, USA
zxh201@psu.edu

Danny Z. Chen

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana, 46556, USA
dchen@nd.edu

Jinhui Xu

Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, New York, 14260, USA
jinhui@buffalo.edu

Abstract

In this paper, we study a generalization of the Voronoi diagram, called the *Influence-based Voronoi Diagram (IVD)*. The input is a collection of possibly overlapping point clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ in some fixed dimensional space $\mathbb{R}^d$ and an influence function $F(C, q)$ measuring the influence from a set C of points to any point q in $\mathbb{R}^d$. The goal is to construct a Voronoi diagram for $\mathcal{C}$ so that each Voronoi cell consists of all points that receive their maximum influence from the same cluster in $\mathcal{C}$. By making use of a recent work called the *Clustering Induced Voronoi Diagram (CIVD)* for unclustered points, we are able to show that it is possible to utilize CIVD's space-partition ability and combine it with a divide-and-conquer algorithm to simultaneously resolve the space partition and assignment problems for a large class of influence functions. This overcomes a major difficulty of CIVD related to its assignment problem. Our technique yields a $(1-\epsilon)$-approximate IVD of size $O(|P|\log|P|)$ in $O(T_2(N)N\log^2 N + T_1(N))$ time, where P is the union of all points of $\mathcal{C}$, N is the total size of all clusters in $\mathcal{C}$, $\epsilon > 0$ is a small constant, and T_1 and T_2 are the functions measuring how

[☆]The research of the first and third authors was partially supported by NSF through grants CCF-1422324 and IIS-1422591; the research of the second author was supported in part by NSF through grant CCF-1617735, and the research of the third author was also supported in part by NSF through grants CCF-1716400 and IIS-1910492.
Corresponding author: Ziyun Huang

efficiently $F(C, q)$ can be evaluated.

Keywords: Voronoi Diagram; Influence Based Voronoi Diagram; Clustering Induced Voronoi Diagram; Influence Function

---

1. Introduction

The Voronoi diagram is a fundamental geometric structure with numerous applications in many different areas. Given a set P of points or objects (called *sites*) in $\mathbb{R}^d$, the (ordinary) Voronoi diagram induced by P is a partition of the space $\mathbb{R}^d$ into a set of cells, where each cell of the diagram is the union of all points in $\mathbb{R}^d$ whose distances to a particular site are closer (or farther) than to any other sites. There are many variants of Voronoi diagrams, depending on the type of objects in P, the distance metric, *etc.* Voronoi diagram can be viewed as the result of a competition among sites in P, where for any point $q \in \mathbb{R}^d$, the winning site for q is the one that has the largest "influence" on q. For most of the known Voronoi diagrams, the influence from each site is independent from the other sites in the sense that the influences from multiple sites are not aggregated. However, as it will be shown later, many real world applications expect that the influence from multiple objects can be combined to form a certain type of *joint influence*.

To accommodate such an expectation, we generalize in this paper the concept of Voronoi diagram to the *Influence-based Voronoi Diagram (IVD)*. Let P be a set of points in $\mathbb{R}^d$ for some fixed integer $d > 0$, and $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ be a collection of (possibly overlapping) clusters of points of P (*i.e.* $C_i \subseteq P, i = 1, 2, \ldots, n$). For any cluster $C \subseteq P$, all the points in C have a combined *joint influence* on every point $q \in \mathbb{R}^d$, which is measured by a non-negative real function $F(C, q)$ (called *influence function*). The cluster $C_i \in \mathcal{C}$ that has the largest joint influence $F(C_i, q)$ on q is called the *Maximum Influence Cluster* (MIC) of q. An IVD of $\mathcal{C}$ is a partition of the space $\mathbb{R}^d$ into regions (called Voronoi cells) such that each Voronoi cell c is associated with a cluster (called Voronoi site) $C_i \in \mathcal{C}$ that is shared by all points c as their common MIC. There are two major differences between IVDs and the traditional Voronoi diagrams (VD). One difference is that each Voronoi site of an IVD is a given point cluster, while the Voronoi site of a traditional VD is one input point. The other difference is that the IVD is influence based, while the traditional VDs are distance based in general. This means that when determining the Voronoi cell for a point q, IVD measures the joint influence from all points in a Voronoi site to q by an influence function, but the traditional VDs compute the distance (*e.g.,* Euclidean or Hausdorff distance) between the Voronoi site and q.

The rationales of generalizing distance-based Voronoi diagrams to influence-based Voronoi diagrams are well justified in [5, 6], as joint influence is a rather common phenomenon in real world (such as in physics, social networks, *etc.*). Chen *et al.* were the first to study the influence based Voronoi diagrams in their work on *Clustering Induced Voronoi Diagrams (CIVD)* [5]. They showed that for

any set P of n points in $\mathbb{R}^d$ and an influence function $F(C, q)$ satisfying some general conditions (*i.e.,* Similarity Invariant, Locality, and Local Domination properties), it is possible to partition the $\mathbb{R}^d$ space into $O(n \log n)$ cells so that all points in each cell share a common subset of P as their approximate MIC. Their CIVD technique first partitions the space into a set of cells (called *partitioning*) based on a general Approximate Influence (AI) Decomposition technique, and then determines the approximate MIC (or AMIC) for each cell by a problem-specific assignment algorithm (called *assignment*).

The CIVD considers all possible subsets (*i.e.,* the power set) of P as its potential Voronoi sites, and the Voronoi sites, as well as the cells, are solely induced by the influence function. This greatly increases the freedom for CIVD to capture the domination relationships between the input points and the space. It also elevates the challenge level of constructing such a diagram. For instance, it is evident in [5] that although a general technique (*i.e.,* AI decomposition, introduced in [5]) exists for a large class of influence functions to partition the space, it is quite unlikely to find a common assignment algorithm for such a class of influence functions (the assignment problem was solved in [5] in a problem-dependent manner).

To address the difficulty associated with the assignment problem, we consider the IVD problem in this paper. In some sense, the IVD model can be viewed as a special case of CIVD, where the Voronoi sites are restricted to be the set $\mathcal{C}$ of given clusters, instead of the power set of the input point set P. We expect that such a restriction can enable us to characterize a set of general conditions (which might be somewhat different from those of the CIVD model) such that for any influence function satisfying this set of conditions, its corresponding IVD can be computed by a common algorithm efficiently, thus simultaneously solving the space partition and the assignment problems.

The IVD model has a number of potential applications. One such application comes from a recent interesting medical study. In [13, 14], Wang *et al.* showed that by constructing the Voronoi diagram of a collection of density-based clusters of already identified (or known) neutrophils (which are a type of white blood cells that help the immune system defend against infections), it is possible to significantly improve the accuracy (by about 10%) of identifying the more difficult candidates of neutrophils (*i.e.,* determining whether a cell that looks somewhat likely to be a neutrophil is indeed a neutrophil) in H&E staining histology tissue images. The ability to identify neutrophils in a large population of immune cells of mixed types in images is critical for the diagnosis of inflamation diseases. Their method suggests that the identification decision for a "suspicious" neutrophil inside the IVD Voronoi cell of a cluster of already identified neutrophils can be based on the "context" of this "suspicious" neutrophil (specified by its Voronoi cell and the neighboring Voronoi cells together with their clusters of already identified neutrophils). The IVD model for neutrophil identification offers the first model of a context for cell studies which allows quantitative analysis [13, 14]. This provides a new way (*i.e.,* using the joint influence of a set of clusters of known neutrophils to identify additional neutrophils) to solve a challenging medical imaging problem.

Another interesting application of the IVD model comes from machine learning. In a recent paper [12], Polianskii *et al.* proposed a space partition technique for classification using Voronoi diagram. Their technique partitions the feature space based on the (weighted) Voronoi Boundary Rank of each class. Mathematically, their space partition can be viewed as a special IVD of a set of nearby classes of feature points, where the influence function is defined as the Voronoi Boundary Rank.

It is worth pointing out that several distance-based Voronoi diagrams actually allow their Voronoi sites to have multiple points. These include the k-th order Voronoi diagrams [10], the Hausdorff Voronoi diagrams [11, 4, 15], and the two-point site Voronoi diagrams [2, 9]. The distance functions used in these works are often defined by the closest (or farthest) point in the Voronoi sites, not by a collective effect of all points of these sites. By studying the IVD model we explore and extend the concept of the Voronoi Diagrams from a different perspective by considering joint influence of a group of points. In fact, some of these distance-based Voronoi diagrams, including the ordinary Voronoi diagram, 2-point site Voronoi diagrams and k-order Voronoi diagrams, can be formulated as IVDs by choosing an appropriate influence function and clusters collections.

Main Results and Techniques. In this paper we show that, for any influence function that satisfies serveral natrual properties (discussed in **Section 2**), a $(1-\epsilon)$-approximate IVD with size $O(|P|\log|P|)$ can be built in $O(T_2(N)N\log^2 N + T_1(N))$ time, where N is total size of the clusters in $\mathcal{C}$, $\epsilon > 0$ is a small constant, and T_1 and T_2 are the functions measuring how efficiently $F(C,q)$ can be evaluated. The hidden constant factor in the big-O notation is $O(1/\Delta(\epsilon)^{O(d)})$, where $\Delta(\cdot) < 1$ is a function depending on $F(C,q)$. Our technique relies on a space partition technique called AI Decomposition from [5] and a novel divide-and-conquer based assignment scheme. The AI Decomposition is used to generate a space partition for the IVD. However, this technique alone cannot solve the assignment problem. Thus, a divide-and-conquer based assignment scheme is proposed. To efficiently implement this assignment scheme, we combine the AI Decomposition algorithm with a tree pruning technique. This resultis in a new quad-tree decomposition algorithm called *Assisted AI Decomposition*.

2. Problem Description and Technique Overview

In this section, we give an overview of our approach for building an approximate IVD of a given set of point clusters.

Definition 1. Let P be a set of points in $\mathbb{R}^d$, and $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ be a set of clusters of points of P with $P = \bigcup_{C_i \in \mathcal{C}} C_i$, and $F(C,q)$ be a nonnegative function (called the *influence function*) for a point cluster $C \subset \mathbb{R}^d$ to any query point q in $\mathbb{R}^d$ space. For a small constant $\epsilon > 0$, the $(1-\epsilon)$-*approximate IVD of* $\mathcal{C}$ *(denoted by* $\mathsf{AIVD}(\mathcal{C})$*)* is a partition of $\mathbb{R}^d$ into cells $\{c_1, c_2, \ldots, c_t\}$ such that each cell c_j is associated with a cluster $C(c_j) \in \mathcal{C}$ satisfying the condition $F(C(c_j), q) \geq (1-\epsilon)F(C_k, q)$ for all $q \in c_j$ and $k \in \{1, 2, \ldots, n\}$.

Figure 1 shows an example of an approximate IVD of 3 clusters with influence function $F(C, q) = \sum_{p \in C} 1/\|p - q\|_2$.

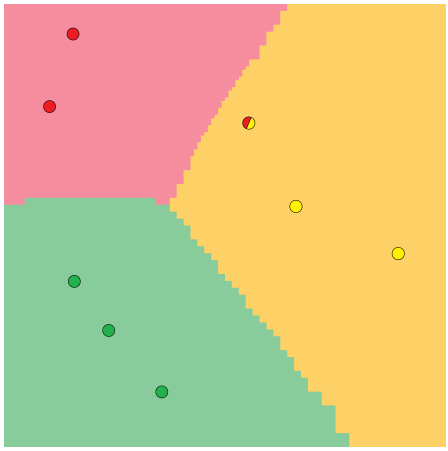

Figure 1: An example of an approximated IVD of three clusters, each of which consists of three input points. In this example, two clusters (the red and yellow) share a input point (marked as yellow and red). The influence function is $F(C, q) = \sum_{p \in C} 1/\|p - q\|_2$ and ϵ is set to be 0.1.

In the rest of this paper, we let N be the total size of all clusters in $\mathcal{C}$, *i.e.,* $N = \sum_{C_i \in \mathcal{C}} |C_i|$.

In the above definition, $C(c_j)$ is called the approximate maximum influence cluster (AMIC) of c_j. The Voronoi region of a cluster may either be empty or consist of one or more (possibly disjoint) cells. This means that the total number of cells could be much larger (*e.g.,* exponentially larger) than the number of clusters (*i.e.,* $t \gg n$). To ensure that the yielded Voronoi diagram has a small size, we expect the influence function $F(C, q)$ to have the following properties, which are slight modifications from those in [5].

(P.1) **Invariance Under Translation:** $F(C, q)$ remains unchanged if C and q are translated by a same vector in $\mathbb{R}^d$.

(P.2) **Majority Rule:** If all points in C coincide at a common location, adding another coincident point to C does not decrease $F(C, q)$.

(P.3) **Locality:** If every point $p \in C$ is perturbed by a relative distance $\Delta(\epsilon)$ to q (*i.e.,* $||p - new(p)|| \leq \Delta(\epsilon)||p - q||$ for some positive function $\Delta(\cdot) < 1$), $F(C, q)$ changes by a factor no more than ϵ.

(P.4) **Local Domination:** There exists a polynomially bounded function $\mathcal{P}(\cdot)$ such that for sufficiently small $\epsilon > 0$, the following holds: For any $C' \subset C$, if there is a point $p \in C'$ with $\mathcal{P}(|C|)\|q - p\| \leq \epsilon \cdot \|q - p'\|$ for all $p' \in C \setminus C'$, then $(1 - \epsilon)F(C, q) \leq F(C', q) \leq (1 + \epsilon)F(C, q)$.

(P.5) **Efficient Estimation:** An approximate value of $F(C, q)$ can be estimated efficiently. A data structure of $\mathcal{C}$ can be built in $T_1(N)$ time and outputs $(1\pm\epsilon)$-approximate value of $F(C_i, q)$ in $T_2(N)$ time for any $C_i \in \mathcal{C}$, $q \in \mathbb{R}^d$, and small $\epsilon > 0$.

To build the $\mathsf{AIVD}(\mathcal{C})$, our main tasks are to decompose $\mathbb{R}^d$ space into cells $\{c_1, c_2, \ldots, c_t\}$ and assign to each cell c_j its approximate maximum influence cluster $C(c_j)$ from $\mathcal{C}$ (this procedure is called the *assignment* of c_j). For the space decomposition, our idea is to make use of the *Approximate Influence (AI)* decomposition technique developed for the CIVD problem [5], which partitions the $\mathbb{R}^d$ space into $O(|P| \log |P|)$ cells for a set of unclustered points P. Since AI decomposition considers all possible subsets of P (*i.e.,* the power set 2^P), it certainly takes all clusters in $\mathcal{C}$ into consideration. Briefly speaking, the space partition generated by the AI decomposition ensures that, for every subset C of P (and thus for every C from $\mathcal{C}$), most of generated cells c satisfies that all points in c receives similar influence from C. This implies that the resulting cells of AI decomposition can be used to form the partition of $\mathsf{AIVD}(\mathcal{C})$. However, to achieve this goal, we still need to solve two main problems. (1) Identify and merge smaller cells in the AI decomposition to form larger cells for clusters in $\mathcal{C}$ (we call this the simplification of the partition of AI decomposition). This is because a cell of $\mathsf{AIVD}(\mathcal{C})$ associated with a cluster $C_i \in \mathcal{C}$ could be further partitioned into a number of smaller cells in the AI decomposition (due to the fact that AI decomposition considers all subsets of C_i). (2) Assign a cluster in $\mathcal{C}$ to each of these merged cells (since AI decomposition does not solve the assignment problem). To simultaneously solve the two problems we develop a divide-and-conquer approach, which has the following main steps.

1. **Partitioning:** Use an Assisted AI (*i.e.,* a modified AI) decomposition on all points in $P = \bigcup_{C \in \mathcal{C}} C$ to partition the space into a set of cells (also denoted as $\mathsf{AIVD}(\mathcal{C})$) *with no assignments.*
2. **Dividing:** Divide $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ into two subsets, $\mathcal{C}_1 = \{C_{11}, C_{12}, \ldots, C_{1g}\}$ and $\mathcal{C}_2 = \{C_{21}, C_{22}, \ldots, C_{2h}\}$.
3. **Recursing:** Recursively build $\mathsf{AIVD}(\mathcal{C}_1)$ and $\mathsf{AIVD}(\mathcal{C}_2)$ *with assignment.*
4. **Merging:** Merge $\mathsf{AIVD}(\mathcal{C}_1)$ and $\mathsf{AIVD}(\mathcal{C}_2)$ to obtain $\mathsf{AIVD}(\mathcal{C})$ with assignments.

To implement the above approach, one of the main difficulties is how to control the errors within the desired range. This is because approximation error are incurred in each **Merging** step and could accumulate throughout the recursion (*i.e.,* in the **Recursing** step). To overcome this difficulty, our idea is to maintain the following **Containing Condition** in every **Merging** step: for each cell c of $\mathsf{AIVD}(\mathcal{C})$, there exist a cell $c_1 \in \mathsf{AIVD}(\mathcal{C}_1)$ and a cell $c_2 \in \mathsf{AIVD}(\mathcal{C}_2))$ both containing c (*i.e.,* the regions of c_1 and c_2 containing the region of c). With the help of this condition, we are able to show (through a rather involving analysis) that the accumulative error is no more than the error tolerance.

Of course, to maintain the containing condition is itself a challenge. Our strategy is to first generate a simplification of $\mathsf{AIVD}(\mathcal{C})$ for each of the two subsets $\mathcal{C}_1$ and $\mathcal{C}_2$. Particularly, we extend the AI decomposition to an *Assisted AI Decomposition*. This allows us to produce a pruned version (namely $\mathsf{AIVD}(\mathcal{C}_1)$ and $\mathsf{AIVD}(\mathcal{C}_2)$) of $\mathsf{AIVD}(\mathcal{C})$ satisfying the containing condition for each of the two subsets $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively(see Figure 2). The imposed containing condition on the pruned versions of the decomposition enables us to recursively relate cells of $\mathsf{AIVD}(\mathcal{C})$ to cells of $\mathsf{AIVD}(\mathcal{C}_1)$ and $\mathsf{AIVD}(\mathcal{C}_2)$ and simplify the assignment process for all cells in $\mathsf{AIVD}(\mathcal{C})$.

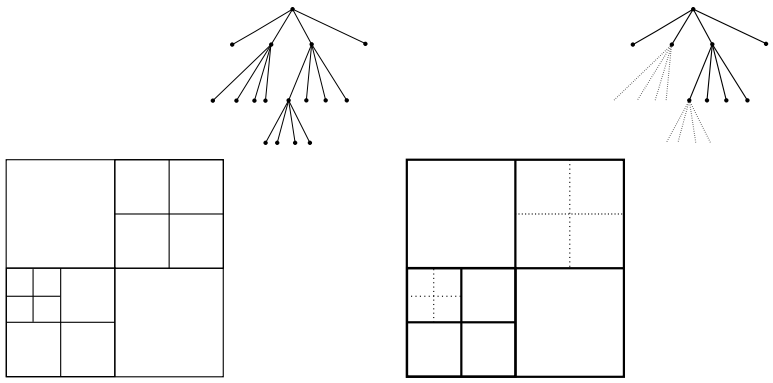

Figure 2: An example showing two VDs and their quad-trees. The right side one is a pruned version of the left side one. The dashed lines indicate the regions that are pruned.

To analyze the proposed approach, we provide a correctness proof and a complexity analysis. The analysis shows that our approach yields an $\mathsf{AIVD}(\mathcal{C})$ with size $O(|P|\log|P|)$ in $O(T_2(N)N\log^2 N + T_1(N))$ time (recall that N is the total size of clusters in $\mathcal{C}$), where the hidden constant factor in the big-O notation is $O(1/\Delta(\epsilon)^{O(d)})$, and $\Delta(\cdot) < 1$ is a function depending on the influence function.

3. Assisted AI Decomposition

In this section, we show how the AI decomposition can be extended for our AIVD problem. For the sake of completeness, we start with a brief overview of the AI decomposition [5]. A more detailed description of AI Decomposition can be found in See **Section 6.1**.

3.1. A Brief Overview of the AI Decomposition

Let P be a set of points in $\mathbb{R}^d$ space, and $F(C, q)$ be an influence function measuring the influence from any subset C of P to an arbitrary point $q \in \mathbb{R}^d$. AI decomposition is a recursive quad-tree decomposition scheme. It produces a data structure called the *box-tree* for a set of unclustered points P, where every

node (called a *box-node*) is a region in $\mathbb{R}^d$ space. Each region c is either a box (*i.e.,* an axis-aligned hypercube) or the difference of two boxes, and all children of every non-leaf node of the box-tree form a partition of this node's region. To reduce the number of subsets that need to be tracked, AI decomposition adopts a strategy that views multiple input points as a single "heavy" point, and relies on a key data structure called the *distance-tree* [5] to implement it. (See **Section 6.1** for a detailed description of distance tree.) Each node (called a *distance-node*) of the binary-tree-structured distance-tree corresponds to a subset of input points that can be viewed as a heavy point.

The regions associated with leaf nodes (called *cells*) of the box-tree form the partition of $\mathbb{R}^d$, where each cell c corresponds to a (possibly unknown) subset C which is a $(1-\epsilon)$-approximate maximum influence set of all points in c. AI decomposition generates two types of cells, type-1 and type-2 cells. A type-1 cell is a region very close to a known subset C of P (compared to points in $P \setminus C$) and the diameter of C is small enough compared to its distance to the cell; in this case, the cell is said to be **dominated** by C. A type-2 cell is a region which is small enough compared to its distance to any point in P (see **Figure 5** in Section **6.1** for an illustration of the two types of cells). Note that most generated cells are boxes, while some of the type-1 cells could be difference of 2 boxes. See **Algorithm 8** in **Section 6.1** for more details.

The AMIC of a type-1 cell is its (known) dominating set, while the AMIC of a type-2 cell is unknown in general and needs to be determined from the power set of P.

Note that there were previous results on approximate Voronoi diagrams which also use quad-tree based space decomposition schemes [7, 1]. These methods are designed mainly for traditional Voronoi diagrams. The cells in these diagrams are induced by approximated closest distance to input points. While in AI decomposition, cells are generated based on influence from all possible subsets of the input points. Thus, the design of AI decomposition is very different from these methods despite that they seem to share some similarity (*e.g.* quad-tree decomposition).

To understand why the partition of AI decomposition is useful for generating AIVD of $\mathcal{C}$, recall that the influence function $F(C, q)$ satisfies properties (P.1)–(P.5). For a type-2 cell c, by (P.1) and (P.3), we know that we may choose an arbitrary point $q_c \in c$ to represent all points in c, since $F(C_i, q')$ differs from $F(C_i, q_c)$ by a little for any $q' \in c$. Thus, all points in c share a common AMIC C_i. For a type-1 cell c dominated by a point set $P' \subseteq P$, by (P.2) and (P.4), we know that C_i that has the most points in P' among $\{C_1, C_2, \ldots, C_n\}$ should be an AMIC for all points in c. This means that the cells generated by AI decomposition can be used to derive the partition of AIVD($\mathcal{C}$).

3.2. Assisted AI Decomposition for Pruning Box-Tree

In this subsection, we show how to modify AI decomposition to make the divide and conquer approach (in Section 2) possible.

Let $\mathcal{C}' = \{C_1, C_2, \ldots, C_m\} \subseteq \mathcal{C}$ be a subset of $\mathcal{C}$. As mentioned in Section 2, we need to generate a box-tree $\mathcal{T}'$ for $\mathcal{C}'$ by simplifying the box-tree $\mathcal{T}_a$ of all

points in $\mathcal{C}$. Particularly, for every cell (*i.e.,* leaf node) c of $\mathcal{T}_a$, we expect that there is a cell c' in $\mathcal{T}'$ wholly containing c. Furthermore, each cell in $\mathcal{T}'$ should be either a type-1 or type-2 cell for points in $\mathcal{C}' = \{C_1, C_2, \ldots, C_m\}$.

To build the box-tree $\mathcal{T}'$, our main idea is to prune $\mathcal{T}_a$. This is implemented by recursively traversing $\mathcal{T}_a$. We start with the root u of $\mathcal{T}_a$, keep u as a node in $\mathcal{T}'$, and recurse on u's children. If u meets the criteria of being a type-1 or type-2 cell for $\mathcal{C}'$ or is already a leaf node in $\mathcal{T}_a$, we stop the recursion on u's children and make u a leaf node of $\mathcal{T}'$ (*i.e., u*'s descendants in $\mathcal{T}_a$ are all removed); otherwise, we continue the traversal recursively on u's children. The type of a cell is verified by the same method used in the AI Decomposition; this means that a distance tree is also built for P in advance (see **Algorithm 7** in **Section 6.1**). The above strategy is made precise in the procedure AssistedDecomposition (Algorithm 1) which is used to determine whether a node of $\mathcal{T}_a$ can be a type-1 or 2 cell for $\mathcal{C}'$.

Note that in the above recursive procedure, $\mathcal{T}_a$ is not actually changed by pruning. Instead, each node u of the newly constructed tree $\mathcal{T}'$ is actually a copy of the node ref(u) of $\mathcal{T}_a$ referring to the same region $B(u)$ in $\mathbb{R}^d$. For convenience, we do not distinguish u and ref(u) in our discussion.

To implement the above pruning approach, a few subtle and yet important modifications are needed to ensure the resulting pruned box-tree satisfies the required containing condition (see Section 2). One subtle modification is on the way of handling type-1 cells. In AI decomposition, if the region $B(u)$ of a box-node u is detected to be very close to a subset $P(v)$ associated with a distance-node v, a new box-node B' is created to represent the region of $E(v) \cap B(u)$, where $E(v)$ is a square region which is considered to be too close to $P(v)$ comparing to the diameter of $P(v)$ (See Algorithm 7 in **Section 6.1** for definition of $E(v)$); the region of $B(u) \setminus B'$ becomes a type-1 cell since it is close to $P(v)$ (comparing to all points in $P \setminus P(v)$) and yet the diameter of $P(v)$ is small enough comparing to its distance to the cell. Since points in B' could have not large enough distance to $P(v)$ comparing to the diameter of $P(v)$, AI decomposition then recursively processes B'. However, in our Assisted AI decomposition, we do not simply treat the region of $B(u) \setminus B'$ as a type-1 cell, for the purpose of making sure that the resulting $\mathcal{T}'$ satisfy the containing condition for pruned box-tree. Instead, we treat those children of B which do not intersect $E(v) \cap B(u)$ as type-1 cells and recursively prune all other children. See Figure 3 for an example. The details of this are given in the algorithm HandleType1Cell (Algorithm 2), a subroutine of the recursive routine AssistedDecomposition (Algorithm 1).

The above modification implies another subtle modification on handling type-1 cells. The Assisted AI decomposition fully simplifies type-2 cells (*i.e.,* once a box-node u is determined as a type-2 cell of $\mathcal{C}'$, all its children are pruned). However, as we discussed above, for a type-1 cell, Assisted AI decomposition actually does not fully simplify it. We will show later that this is sufficient for our approach.

The full Assisted AI Decomposition algorithm is given in the following **Algorithm 4**. The core part is **Algorithm 1**, the procedure to recursively prune the box-tree $\mathcal{T}_a$ to obtain a new box-tree. The $\mathcal{P}(\cdot)$ function appears in step 4

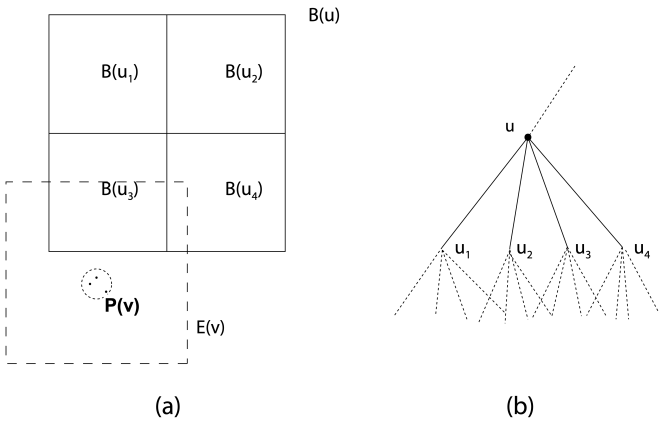

Figure 3: An example showing part of a to-be-pruned tree (b) and the corresponding boxe-nodes and input points (a). $B(u)$ is very close to a set of three points $P(v)$ (the three points in the dashed circle), compared to other input points (not shown in the figure), where v is some distance-node. Then $B(u_1)$ and $B(u_2)$ can be output as type-1 cells in the pruned tree, since they are very close to $P(v)$ compared to other input points, and yet also far enough (outside $E(v)$) to view $P(v)$ as one point. $B(u_3)$ and $B(u_4)$ need further examination/recursion.

of **Algorithm 1** is the polynomially bounded function $\mathcal{P}$ in Property (P.4) of the influence function. The parameter $\beta, 0 < \beta < 1/2$, is an error factor. The subroutine, SearchTail (**Algorithm 3**), is designed to avoid generating long chains in the newly constructed box-tree. For example, if for a sequence of box-nodes $u_1, u_2, \ldots, u_k$, u_{i+1} is the only child of u_i for every i, then we may simply make u_k the only child of u_1 and discard $u_2, u_3, \ldots, u_{k-1}$, for space saving. The algorithm share some similar ideas with the standard quad-tree compression operation (see [8] for an introduction).

Algorithm 1 AssistedDecomposition($u, \beta, L, T_p, r_c, \mathcal{T}_a$)

Input: A box-node u with box $B(u)$, error tolerance $\beta > 0$, a linked list L for storing distance-nodes that need to be examined, a value r_c to measure the estimated closest distance from $B(u)$ to points not in L, a distance-tree T_p, and a box tree $\mathcal{T}_a$.

Output: A subtree of a pruned box-tree $\mathcal{T}_q$ rooted at u.

1: **While** $\exists$ v in L such that the length of at least one edge of $B(u) \cap E(v)$ is no smaller than $\frac{size(B(u))}{2}$ **do**

- Replace v in L by its two children in T_p, if any.

2: Let $D(u)$ be the diameter of $B(u)$. For each node v in L **do**

2.1 Let r_{min} be the distance between $B(u)$ and $l(v)$, where $l(v) \in P(v)$ is a representative point of $P(v)$. (It is generated for v when building the distance-tree. See Algorithm 7 in Section 6.1 for more details.)

2.2 If $D(u) < r_{min}\beta/2$, remove v from L, and if $r_c > r_{min}$, let $r_c = r_{min}$.

3: If L is empty, return, and u is labeled as a **type-2** cell.

4: If there is only one element v in L, let r_{min} be the smallest distance between $l(v)$ and $B(u)$.

4.1 If $\frac{r_{min}+D(u)}{r_c} < \frac{\beta}{2\mathcal{P}(n)}$,

4.1.1 If $E(v) \cap B(u) = \emptyset$ or v is a leaf node in T_p, u is a **type-1** cell dominated by v. Return.

4.1.2 Call HandleType1Cell($u, \beta, v, T_p, r_c, \mathcal{T}_a$) and return.

5: For every child u' of ref(u), **do**

5.1 If u' is type-2 cell in $\mathcal{T}_a$

5.1.1 Create box-node u_c with corresponding box $B(u_c) = B(u')$, and set ref(u_c) $= u'$.

5.1.2 u_c becomes a child of u. And $B(u_c)$ is labeled as a type-2 cell.

5.2 If u' is neither a type-1 cell nor a type-2 cell in $\mathcal{T}_a$, **do**

5.2.1 Create box-node u_c with corresponding box $B(u_c) = B(u')$, and set ref(u_c) $= u'$.

5.2.2 u_c becomes a child of u.

5.2.3 Call AssistedDecomposition($u_c, \beta, L, T_p, r_c, \mathcal{T}_a$)

(Note: If u' is a type-1 cell in $\mathcal{T}_a$, do nothing.)

Algorithm 2 HandleType1Cell($u, \beta, v, T_p, r_c, \mathcal{T}_a$)

Input: A box-node u with box $B(u)$, error tolerance $\beta > 0$, a distance node v from T_p, a value r_c, distance-tree T_p, and a box-tree $\mathcal{T}_a$.
Output: A subtree of a pruned box-tree $\mathcal{T}_q$ rooted at u.
Note: u corresponds to a box-node ref(u) in $\mathcal{T}_a$.

1: If ref(u) is a type-1 or type-2 cell of $\mathcal{T}_a$, u is labeled as a type-1 cell dominated by v or type-2 cell, respectively. Return.

2: If the length of at least one edge of $B(u) \cap E(v)$ is no smaller than $\frac{size(B(u))}{2}$ **do**

- Call AssistedDecomposition($u, \beta, \{v_1, v_2\}, T_p, r_c, \mathcal{T}_a$), where v_1, v_2 are the two children of v in T_p. Return.

3: If no child u' of ref(u) in $\mathcal{T}_a$ satisfies that $B(u') \cap E(v) \neq \emptyset$ and u' is not a type-1 cell in $\mathcal{T}_a$, return and u become a type-1 cell dominated by v.

4: If more than one child of ref(u) in $\mathcal{T}_a$, say $u'_1, u'_2, \ldots, u'_k$, satisfy that $B(u'_i) \cap E(v) \neq \emptyset$ and u'_i is not a type-1 cell in $\mathcal{T}_a$, $i = 1, 2, \ldots, k$, **do the following and then return.**

4.1 For every $u'_i, i = 1, 2, \ldots, k$, **do**

4.1.1 Create box-node u_i, u_i becomes a child of u.

4.1.2 Set ref(u_i) = u'_i, $B(u_i) = B(u'_i)$.

4.1.3 Call HandleType1Cell($u_i, \beta, v, T_p, r_c, \mathcal{T}_a$).

4.2 For every child u' of ref(u) that satisfies $B(u') \cap E(v) = \emptyset$ or u'_i is a type-1 cell in $\mathcal{T}_a$, **do**

4.2.1 Create box-node u_c, u_c becomes a child of u.

4.2.2 Set ref(u_c) = u', $B(u_c) = B(u')$.

4.2.3 u_c is labeled as a type-1 cell.

5: If exactly one child u' of ref(u) in $\mathcal{T}_a$ satisfies $B(u'') \cap E(v) \neq \emptyset$ and u' is not a type-1 cell in $\mathcal{T}_a$, **do**

5.1 Set u_t = SearchTail($\mathcal{T}_a, u', E(v)$).

5.2 Construct 2 box-nodes, u_1 and u_2, as the children of u.

5.3 Set ref(u_1) = u_t and $B(u_1) = B(u_t)$.

5.4 Set $B(u_2)$ as the difference of $B(u)$ and $B(u_1)$. u_2 is labeled as a type-1 cell dominated by v.

5.5 Repeat Step 1 – Step 4 with u set to be u_1.

Algorithm 3 SearchTail($\mathcal{T}_a, u', E(v)$)

Input: A box-tree $\mathcal{T}_a$, a node u' of $\mathcal{T}_a$ and a box $E(v)$.
Output: A node of $\mathcal{T}_a$ which is a the end node of a chain in $\mathcal{T}_a$ starting at u'.

1: Repeat the following until **break**:

- If u' is a type-1 or type-2 cell, **break**.
- If the length of at least one edge of $B(u') \cap E(v)$ is no smaller than $\frac{size(B(u'))}{2}$, **break**.
- If more than one child of u' in $\mathcal{T}_a$, say $u'_1, u'_2, \ldots, u'_k$, satisfy that $B(u'_i) \cap E(v) \neq \emptyset$ and u'_i is not a type-1 cell of $\mathcal{T}_a$, $i = 1, 2, \ldots, k$, **break**.
- If exact one child u'' of u' satisfies that $B(u'') \cap E(v) \neq \emptyset$ and u'' is not a type-1 cell, set $u' = u''$.
- If no child u'' of u' satisfies that $B(u'') \cap E(v) \neq \emptyset$ and u'' is not a type-1 cell, **break**.

2: Return u'.

Algorithm 4 AssistedAIDecomposition($P, \beta, \mathcal{T}_a$)

Input: A set of points P, error tolerance $\beta > 0$ and a box-tree $\mathcal{T}_a$.
Output: A box tree $\mathcal{T}_q$ which simplifies $\mathcal{T}_a$.

1: Build a distance-tree T_p for P (See Algorithm 7 in Section 6.1).
2: Create box-tree node u. Set $B(u)$ as $B(u_r)$ where u_r is the root of $\mathcal{T}_a$. Set ref(u) as u_r.
3: Run AssistedDecomposition($u, \beta, \emptyset, T_p, \infty, \mathcal{T}_a$)

4. The Full Algorithm

The Assisted AI decomposition presented in last section enables us to deal with the Partitioning step of our divide-and-conquer approach. In this section, we focus on the Merging step, as well as the whole algorithm, since the Dividing and Recursing steps are quite straightforward (*e.g.*, dividing the set of clusters into two subsets with nearly equal total cardinality and recursively build the AIVD for each subset). Combining the Assisted AI decomposition and the idea of maintaining an approximated value of the maximum influence of a sub-collection of clusters, the error accumulated during the recursion can be controlled (to be shown in next section).

Let $\mathcal{T}$ be the box-tree generated by the Assisted AI decomposition for the set of clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$. To complete the construction of $\mathsf{AIVD}(\mathcal{C})$, we still need to assign an AMIC to each cell in $\mathcal{T}$ (*i.e.*, the Merging step). As discussed in Section 2, this is done by divide-and-conquer approach. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the simplified box-trees (by Assisted AI decomposition) for the two subsets $\mathcal{C}_1 = \{C_1, C_2, \ldots, C_m\}$ and $\mathcal{C}_2 = \{C_{m+1}, C_{m+2}, \ldots, C_n\}$, respectively, satisfying the containing condition. Since in our divide-and-conquer algorithm, $\mathcal{T}_1$ and $\mathcal{T}_2$ are constructed recursively, we assume that for each cell c of $\mathcal{T}_1$ (or $\mathcal{T}_2$), its AMIC $C(c)$ in $\mathcal{C}_1$ (or $\mathcal{C}_2$) has already been determined. Note that this can be recursively ensured. At the bottom level of recursion where each of $\mathcal{C}_1$ and $\mathcal{C}_2$ has only $O(1)$ clusters, the AMIC of each cell in the corresponding box-tree can be easily determined by computing the influence from each of the $O(1)$ clusters to some arbitrary query point q in the cell and choosing as AMIC the one with the largest influence. Also for each cell c of $\mathcal{T}_1$ (or $\mathcal{T}_2$), we assume that an approximate value $f(c)$ of $F(C(c), q)$ for all query points $q \in c$ has been computed if c is a type-2 cell. Note that such a value can be easily obtained by choosing any query point $q \in c$ and computing the value of $F(C(c), q)$. By the Locality property (P.3) and the definition of type-2 cells, we know that the value of $F(C(c), q)$ differs only a little for all points q in c.

To determine the AMIC for each cell c of $\mathcal{T}$ in $\mathcal{C}$, we first consider its type. If c is a type-1 cell dominated by $P(v)$ for some distance-node v (which is determined by the Assisted AI decomposition when constructing $\mathcal{T}$), the AMIC $C(c)$ of c can be chosen as the cluster in $\mathcal{C}$ which has the maximum number of points in $P(v)$ (*i.e.,* $C(c)$ is the cluster maximizing the size $|C_i \cap P(v)|$ for all $C_i \in \mathcal{C}$). This is because by the definition of type-1 cell, we know that the influence from all points in $P \setminus P(v)$ to any point $q \in c$ can be neglected; by Locality property (P.3), we can view all points in $P(v)$ as a single heavy point, say p, (*i.e.,* all points coincident at p); and by Majority Rule (P.2), we know that the cluster $C(c)$ has the largest influence on q.

If c is a type-2 cell, since $\mathcal{T}_1$ and $\mathcal{T}_2$ simplify $\mathcal{T}$, we can find a cell c_1 from $\mathcal{T}_1$ and a cell c_2 from $\mathcal{T}_2$ such that both c_1 and c_2 wholly contain c (by the Containing Condition). If c_j, for $j = 1, 2$, is a type-2 cell, we also have an approximate value, say $f(c_j)$, for all $q \in c_j$; otherwise (*i.e.,* c_j is a type-1 cell), we can pick any point $q \in c$, and compute an approximate value of $F(C(c_j), q)$ as $f(c_j)$. By the assumption, we know that an AMIC $C(c_j) \in \mathcal{C}_j$ has already

been obtained for c_j. Thus, the AMIC $C(c)$ of c can be chosen from $C(c_1)$ and $C(c_2)$ by comparing their corresponding approximate values of $f(c_1)$ and $f(c_2)$ (*i.e.,* choose the one with a larger value). The approximate value $f(c)$ of $F(C(c), q)$ in c can be chosen as the larger one of $f(c_1)$ and $f(c_2)$.

The details of the full algorithm for building $\mathsf{AIVD}(\mathcal{C})$ are given in **Algorithm 5**. The two parameters $\Delta_1(\epsilon)$ and $\Delta_2(\epsilon)$, which depend on ϵ, will be discussed in the later analysis section. Note that, if $\mathcal{C}$ has no more than three clusters, we directly compute the assignment for all cells in its box-tree (see step 3 of **Algorithm 5**). This is the stopping condition for our recursive approach.

Algorithm 5 BuildAIVD($\mathcal{C}, \epsilon, \mathcal{T}_a$)

Input: A Collection $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ of set of input points in $\mathbb{R}^d$. Error tolerance parameter $0 < \epsilon < 1$. A box tree $\mathcal{T}_a$ from upper recursion level to assist AI decomposition.

Output: A box-tree $\mathcal{T}$ which simplifies $\mathcal{T}_p$. An assignment of $(1-\epsilon)$-approximate maximum influence cluster $C(c)$ in $\mathcal{C}$ for each cell c of $\mathcal{T}$
A value val(c) for every type-2 cell c of $\mathcal{T}$.

1: Let $P = \bigcup_{C \in \mathcal{C}} C$. Let $0 < \beta < 1/2$ be a positive number that satisfies $\frac{1+\delta}{1-\delta} \cdot \frac{1+\beta}{1-\beta} \leq (1 - \Delta(\epsilon))^{-1}$, $\beta < \Delta(\epsilon)$ and $\beta < \Delta(\delta)$ for some $0 < \delta < 1$, where $\Delta(\cdot)$ is the function ensured by condition (P.3) Do the following to build the box-tree $\mathcal{T}$

  1.1 If $\mathcal{T}_a$ is **NULL**, Call AIDecomposition(P, β)

  1.2 If $\mathcal{T}_a$ is not **NULL**, Call AssistedAIDecomposition($P, \beta, \mathcal{T}_a$)

2: For every type-1 cell c of $\mathcal{T}$, set the assignment $C(c) = C_j$ for c, where points of C_j is majority in the distance node that dominates c.

3: If $n \leq 3$, do the following.

  3.1 For every type-2 cell c of $\mathcal{T}$, choose arbitrary point q in c.

    3.1.1 Evaluate $F_e(C_i, q)$ which is an $\Delta_2(\epsilon)$-error estimate of $F(C_i, q)$ for all C_i in $\mathcal{C}$. Assume C_j has the maximum $F_e(C_j, q)$

    3.1.2 Set the assignment $C(c) = C_j$ for c.

    3.1.3 Set val(c) = $F_e(C_j, q)$.

4: If $n > 3$, do the following.

  4.1 Sort clusters in $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ in increasing order of cardinality. letting $\{C_1, C_2, \ldots, C_n\}$ denote the result.

  4.2 Let $C_m \in \mathcal{C}$ be such that $\sum_{i=1}^{m} C_i \geq M/2$ and $\sum_{i=1}^{m-1} C_i < M/2$, where $M = \sum_{i=1}^{n} C_i$. Divide $\mathcal{C}$ into $\mathcal{C}_1 = \{C_1, C_2, \ldots, C_m\}$ and $\mathcal{C}_2 = \{C_{m+1}, C_{m+2}, \ldots, C_n\}$.

  4.3 Call BuildAIVD($\mathcal{C}_1, \epsilon, \mathcal{T}$) and BuildAIVD($\mathcal{C}_2, \epsilon, \mathcal{T}$) to build box-trees $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively.

  4.4 Call Merge($\mathcal{T}, \mathcal{T}_1, \mathcal{T}_2$). Return $\mathcal{T}$ as the result.

Algorithm 6 Merge($\mathcal{T}, \mathcal{T}_1, \mathcal{T}_2$)

Input: Box-tree $\mathcal{T}_1, \mathcal{T}_2$ generated by BuildAIVD, Box-tree $\mathcal{T}$.

Output: An assignment of approximate maximum influence cluster $C(c)$ in $\mathcal{C}$ for each cell c of $\mathcal{T}$. A value val(c) for every type-2 cell c of $\mathcal{T}$ which is an estimation of the value of $F(C(c), q)$ for any point $q \in c$.

1: For $j = 1, 2$ and every cell c in $\mathcal{T}_j$,

1.1 Find all the type-2 cells of $\mathcal{T}$ that are contained in c, *i.e.*, type-2 cell leaves of $\mathcal{T}$ that are descendant of c (if $B(c)$ is a box) or c's parent (if $B(c)$ is the difference of two boxes).

1.2 For each type-2 cell c' found in the previous step,

1.2.1 **If c is a type-1 cell of $\mathcal{T}_j$**, choose any point q from c' compute $F_e(C(c), q)$ which is an $\Delta_2(\epsilon)$-error estimation of $F(C(c), q)$ and set $f_j(c)$ as $F_e(C(c), q)$. Let $C(c, j)$ denote $C(c)$.

1.2.2 **If c is a type-2 cell of $\mathcal{T}_j$**, set $f_j(c)$ as val(c). Let $C(c, j)$ denote $C(c)$.

2: For every type-2 cell c in $\mathcal{T}$

2.1 If $f_1(c) \geq f_2(c)$, set $C(c) = C(c, 1)$ and val(c) $= f_1(c)$.

2.2 Otherwise set $C(c) = C(c, 2)$ and val(c) $= f_2(c)$.

5. Algorithm Analysis

5.1. Proof of Correctness

In this subsection, we show that with carefully chosen parameters, algorithm BuildAIVD will produce an AIVD for any given set of clusters with the desired quality.

The following lemmas from [5] reveal important geometry properties of cells generated by AI Decomposition. They also apply to cells generated by the Assisted AI Decomposition which is a modification to AI Decomposition. We list them here without proof.

Definition 2. A distance-node $v \in T_p$ is said to be *recorded* for a box-node u if v is removed from the list L in Step 2.2 of **Algorithm 1** (or **Algorithm 8**, the AI Decomposition, in **Section 6.1**) when processing u or one of u's ancestors. The value of r_{min} in the iteration when v is removed from L is the *recorded distance* of v for u. If v is recorded for u, then any point $p \in P(v)$ is also *recorded* for u with the same recorded distance as v.

Let β be the error controlling parameter used for Algorithm 1.

Lemma 1. *If $p \in P$ is recorded for a box-node u with a recorded distance x, then for any point $q \in B(u)$,*

$$(1-\beta)x \leq \|p-q\| \leq (1+\beta)x.$$

Lemma 2. *For any type-2 cell c and $p \in P$, let $D(c)$ be the diameter of c and r be the shortest distance between c and p. Then*

$$D(c) \leq \frac{2r\beta}{3}. \tag{1}$$

Lemma 3. *If c is a type-1 cell dominated by a distance-node v, then for any $q \in c$ and $p' \in P \setminus P_v$,*

$$\frac{\|q-l(v)\|}{\|q-p'\|} \leq \frac{\beta}{\mathcal{P}(|P|)}.$$

Since the influence function has property (P.3) for some function $\Delta(\cdot)$, by properties of type-1 and type-2 cells, a bound about the approximate quality of AssistedDecomposition can be obtained by the following theorem.

Theorem 1. *Let c be any cell generated by the AsistedAIDecomposition (**Algorithm 4**) or AIDecomposition (**Algorithm 9**, in* **Section 6.1**.*) with an error tolerance $0 < \beta < 1/2$ satisfies $\frac{1+\delta}{1-\delta} \cdot \frac{1+\beta}{1-\beta} \leq (1-\epsilon)^{-1}$, $\beta < \Delta(\epsilon)$ and $\beta < \Delta(\delta)$for some $0 < \delta < 1$, where Δ is the function ensured by property (P.3). Then the following holds.*

1. *If c is a type-1 cell dominated by a distance-node v, then $F(C_m, q) \geq (1-\epsilon)F(C_i, q)$ for query point q in c and every $C_i \in \mathcal{C}$, where C_m is the cluster in $\mathcal{C}$ which has the most number of points in v, and q, in the case that c is generated by **Algorithm 4**, is not contained in a type-1 cell of $\mathcal{T}_a$.*

2. *If c is a type-2 cell and q and q' are two arbitrary points in c, then* $(1-\epsilon)F(C,q) \leq F(C,q') \leq (1+\epsilon)F(C,q)$ *for any* $C \in \mathcal{C}$

PROOF. For case 1 of the theorem, we define a mapping ψ_1 on P as follows.

$$\psi_1(p) = \begin{cases} p & \text{if } p \notin P(v), \\ l(v) & \text{if } p \in P(v). \end{cases}$$

Note that $\psi_1(P) = \psi_1(P(v)) \cup \psi_1(P \setminus P(v))$ ($\psi_1(\cdot)$ is a multiset). By **Algorithm 1**, q cannot be in $E(v)$, thus $\|l(v) - q\| \geq \frac{4s(v)}{\beta}$. For any $p \in P$, clearly $\|p - \psi_1(p)\| \leq s(v)$ since $s(v)$ place an upper bound on diameter of $P(v)$. Therefore, $\frac{\|p-\psi_1(p)\|}{\|l(v)-q\|} \leq \beta$ for any $p \in P$. Since by definition of $\psi_1(\cdot)$ either $\psi_1(p) - p = 0$ or $\psi_1(p) = l(v)$. In both cases

$$\frac{\|p - \psi_1(p)\|}{\|\psi_1(p) - q\|} \leq \beta. \tag{2}$$

Let C be any cluster in $\mathcal{C}$. Let $C(v) = P(v) \cap C \subseteq C$. By **Lemma 2**, equation (2), and the assumption $\beta \leq \Delta(\delta)$, we have

$$F(C,q) \leq (1+\delta)F(\psi_1(C), q).$$

By property (P.4) and **Lemma 3**, we get

$$F(\psi_1(C), q) \leq (1-\beta)^{-1} F(\psi_1(C(v)), q).$$

By property (P.2) and **Algorithm 1**, we obtain

$$F(\psi_1(C(v)), q) \leq F(\psi_1(C_c(v)), q),$$

where $C_c(v) = C(c) \cap P(v)$. By property (P.4) and **Lemma 3**, we have

$$F(\psi_1(C_c(v)), q) \leq (1+\beta)F(\psi_1(C(c)), q).$$

Again by **Lemma 2**, equation (2), and the assumption $\beta \leq \Delta(\delta)$, we get

$$F(\psi_1(C(c)), q) \leq (1-\delta)^{-1} F(C(c), q).$$

Combining the above inequalities gives us

$$F(C,q) \leq \frac{1+\delta}{1-\delta} \cdot \frac{1+\beta}{1-\beta} F(C(c), q).$$

Case 1 of the theorem then follows from the assumption $\frac{1+\delta}{1-\delta} \cdot \frac{1+\beta}{1-\beta} \leq (1-\epsilon)^{-1}$.

Now consider the second case of the theorem. Note that by property (P.1), we have $F(C', q) = F(C, q')$, where C' is obtained by shifting every point p in C by vector $q - q'$, *i.e.,* to a new location $\psi_2(p) = p + q - q'$. By **Lemma 2**, we know that for any $p \in C$, $\|p - \psi_2(p)\| = \|q - q'\| \leq D(c) \leq \|q' - p\|\beta$; since $\beta \leq \Delta(\epsilon)$, by property (P.3), we have $(1-\epsilon)F(C,q) \leq F(C',q) \leq (1+\epsilon)F(C,q)$. The theorem follows from $F(C', q) = F(C, q')$. □

From the algorithm and the containing condition, we know that the following is true.

Observation 1. *Let $\mathcal{T}$ be the box-tree generated by AssistedAIDecomposition$(P, \beta, \mathcal{T}_a)$ (**Algorithm 4**). Then for any type-2 cell c in $\mathcal{T}_a$, there exists a cell c' in $\mathcal{T}$, such that $ref(c')$ is an ancestor of c in $\mathcal{T}_a$. In other words, $B(c')$ wholly contains $B(c)$.*

With the above results, the correctness of BuildAIVD is ensured by the following **Theorem 2**. To prove the theorem, we particularly show that the approximation error does not accumulate. Based on the property of AIVD generated by Assisted AI Decomposition, we know that the maintained value of approximated maximum influence from a subset of the given clusters remains valid after a series of merging operations along the path of the recursion. The theorem is then proved by an induction on the recursion process, in a bottom-up manner.

Theorem 2. *For any sufficient small $\epsilon > 0$, the procedure BuildAIVD($\mathcal{C}, \epsilon$, NULL) (**Algorithm 5**) where parameters $\Delta_1(\epsilon) > 0$ and $\Delta_2(\epsilon) > 0$ satisfy $\epsilon > 1 - (1 - \Delta_1(\epsilon))^2(1 + \Delta_1(\epsilon))^{-1}(1 - \Delta_2(\epsilon))(1 + \Delta_2(\epsilon))^{-1}$, $\Delta_1(\epsilon) < \epsilon$ and $\Delta_2(\epsilon) < \epsilon$, generates a box-tree $\mathcal{T}$ such that for any cell c in $\mathcal{T}$ and any point q in c, $F(C(c), q) \geq (1 - \epsilon)F(C, q)$ for any $C \in \mathcal{C}$.*

Proof. If c is a type-1 cell, then by the way that BuildAIVD handles type-1 cells (step 2 of **Algorithm 5**) and **Theorem 1**, we have $F(C(c), q) \geq (1 - \Delta_1(\epsilon))F(C, q)$ for any $C \in \mathcal{C}$. Thus $F(C(c), q) \geq (1 - \Delta_1(\epsilon))F(C, q) \geq (1 - \epsilon)F(C, q)$ for any q in c. Therefore the theorem holds for every type-1 cell c of $\mathcal{T}$.

Now we consider the case that c is a type-2 cell. We will prove the following:

Claim. *For any call to BuildAIVD($\mathcal{C}', \epsilon, \mathcal{T}_a$) which is one of the subsequent calls to BuildAIVD resulting from the initial recursive call to BuildAIVD($\mathcal{C}, \epsilon$,NULL), the generated box-tree $\mathcal{T}'$ satisfies the following: for any type-2 cell c of $\mathcal{T}'$, there exists a value $F_E(C, c)$ for every $C \in \mathcal{C}'$, such that for any point q in c,*

$$(1-\Delta_1(\epsilon))^2(1-\Delta_2(\epsilon))F(C,q) \leq F_E(C,c) \leq (1+\Delta_1(\epsilon))(1+\Delta_2(\epsilon))F(C,q). \quad (3)$$

Furthermore, $val(c) \geq F_E(C, c)$ for any $C \in \mathcal{C}'$, and $val(c) = F_E(C(c), c)$.

Note that if the claim is true, then for any $C \in \mathcal{C}$ and q in c, we have $F(C(c), q) \geq (1 + \Delta_1(\epsilon))^{-1}(1 + \Delta_2(\epsilon))^{-1}F_E(C(c), c) = (1 + \Delta_1(\epsilon))^{-1}(1 + \Delta_2(\epsilon))^{-1}val(c) \geq (1 + \Delta_1(\epsilon))^{-1}(1 + \Delta_2(\epsilon))^{-1}F_E(C, c) \geq (1 - \Delta_1(\epsilon))^2(1 + \Delta_1(\epsilon))^{-1}(1 - \Delta_2(\epsilon))(1 + \Delta_2(\epsilon))^{-1}F(C, q) \geq (1 - \epsilon)F(C, q)$. The theorem then follows.

We prove the claim by induction. For the basis case, we assume that $\mathcal{C}'$ contains no more than three clusters. Then BuildAIVD (**Algorithm 5**) will perform step 3 to directly obtain $C(c)$ and $val(c)$. In step 3, a point q is chosen from

c, and for every $C \in \mathcal{C}'$, $F_e(C,q)$ is computed such that it satisfies the following inequality $(1-\Delta_2(\epsilon))F(C,q) < F_e(C,q) < (1+\Delta_2(\epsilon))F(C,q)$. For an arbitrary point q' from c, by **Theorem 1**, we know that $(1-\Delta_1(\epsilon))F(C,q') < F(C,q) < (1+\Delta_1(\epsilon))F(C,q')$. Therefore, we have $(1-\Delta_1(\epsilon))(1-\Delta_2(\epsilon))F(C,q') \leq F_e(C,q) \leq (1+\Delta_1(\epsilon))(1+\Delta_2(\epsilon))F(C,q')$ for any q' in c. By setting $F_E(C,c)$ to be $F_e(C,q)$, we prove the claim.

Now in the induction step, we assume that $\mathcal{C}'$ contains more than three clusters. As a result of the recursion of the divide-and-conquer approach, $\mathcal{C}'$ is divided into two subsets $\mathcal{C}_1$ and $\mathcal{C}_2$, so that $\mathcal{T}_1$ and $\mathcal{T}_2$ are built recursively by BuildAIVD, respectively. By the induction hypothesis, we can assume that $\mathcal{T}_1$ and $\mathcal{T}_2$ satisfy the claim. We now show that the claim will also hold for $\mathcal{T}'$. Let c be any type-2 cell of $\mathcal{T}'$. Then there exist cells c_1 from $\mathcal{T}_1$ and c_2 from $\mathcal{T}_2$, so that c is contained in c_1 and c_2. For any $C \in \mathcal{C}'$, without loss of generality, we can assume that $C \in \mathcal{C}_1$ and set $F_E(C,c)$ depending on type of c_1 (note that we can argue similarly for the case of $C \in \mathcal{C}_2$).

If c_1 is a type-1 cell of $\mathcal{T}_1$, note that in **Algorithm 6** that merges $\mathcal{T}_1$ and $\mathcal{T}_2$ into $\mathcal{T}'$, $F_e(C(c_1),q)$ which is an approximate value of $F(C(c_1),q)$ for some q in c is computed. We set $F_E(C(c_1),c)$ as $F_e(C(c_1),q)$, and for $C \in \mathcal{C}_1$ other than $C(c_1)$, $F_E(C,c)$ is set to be $\min(F_e(C(c_1),q), F(C,q))$ For $C(c_1)$, the setting of $F_E(C(c_1),c)$ satisfies the condition of Inequality (3). In fact, since c is a type-2 cell of $\mathcal{T}'$ and $C(c_1) \in \mathcal{C}_1 \subset \mathcal{C}'$, by **Theorem 1**, we know that $(1-\Delta_1(\epsilon))F(C(c_1),q') \leq F(C(c_1),q) \leq (1+\Delta_1(\epsilon))F(C(c_1),q')$ for any q' in c. Also $(1-\Delta_2(\epsilon))F(C(c_1),q) \leq F_e(C(c_1),q) \leq (1+\Delta_2(\epsilon))F(C(c_1),q)$; therefore we have

$$\begin{aligned}(1-\Delta_1(\epsilon))(1-\Delta_2(\epsilon))F(C(c_1),q') &\leq F_e(C(c_1),q) \\ &\leq (1+\Delta_1(\epsilon))(1+\Delta_2(\epsilon))F(C(c_1),q')\end{aligned} \tag{4}$$

for any q' in c. This implies Inequality (3). For $C \in \mathcal{C}_1$ other than $C(c_1)$, we can also show that Inequality (3) holds. By definition of $F_E(C,c)$ and **Theorem 1**, we get $F_E(C,c) \leq F(C,q) \leq (1+\Delta_1(\epsilon))F(C,q')$ for any q' in c. Thus the inequality on the right hand side of (3) holds, and we only need to prove the inequality on the left hand side of (3). There are two cases to consider, $F_E(C,c) = F_e(C(c_1),q)$ and $F_E(C,c) = F(C,q)$. In the following, we let q' be an arbitrary point in c. We first consider the case that $F_E(C,c) = F_e(C(c_1),q)$. Note that $F_e(C(c_1),q) \geq (1-\Delta_2(\epsilon))F(C(c_1),q)$, and by **Theorem 1** and that the fact that c_1 is a type-1 cell of $\mathcal{T}_1$, we have $F(C(c_1),q) \geq (1-\Delta_1(\epsilon))F(C,q)$. Again by **Theorem 1**, we get $F(C,q) \geq (1-\Delta_1(\epsilon))F(C,q')$. This means that $F_E(C,c) = F_e(C(c_1),q) \geq (1-\Delta_2(\epsilon))(1-\Delta_1(\epsilon))^2F(C,q')$. Now consider the case that $F_E(C,c) = F(C,q)$. By **Theorem 1**, we know that $F(C,q) \geq (1-\Delta_1(\epsilon))F(C,q')$. Thus $F(C,q) \geq (1-\Delta_1(\epsilon))F(C,q') \geq (1-\Delta_2(\epsilon))(1-\Delta_1(\epsilon))^2F(C,q')$. The inequality on the left hand side of (3) is proved for C.

If c_1 is a type-2 cell of $\mathcal{T}_2$, then we simply set $F_E(C,c)$ to be $F_E(C,c_1)$. Since c_1 wholly contains c, Inequality (3) holds for all q in c because it already holds for all q in c_1.

We now show that the claim is correct by setting the value of $F_E(C,c)$ as above for a type-2 cell c and a cluster $C \in \mathcal{C}'$. If we can show that in

Algorithm 6, $F_E(C(c,j),c) \geq F_E(C,c)$ holds for $j = 1, 2$ and all $C \in \mathcal{C}_j$, then the claim follows. In fact, assume that this is the case, note that $F_E(C(c,1),c)$ and $F_E(C(c,2),c)$ (written as $f_1(c)$ and $f_2(c)$ in **Algorithm 6**) are compared to obtain $C(c)$ and $val(c)$. If $F_E(C(c,1),c) \geq F_E(C(c,2),c)$, then $C(c)$ is set to be $C(c,1)$ and $val(c)$ is set to be $F_E(C(c,1),c)$. Therefore $val(c) = F_E(C(c,1),c) \geq F_E(C,c)$ for all C in $\mathcal{C}_1$ and $val(c) = F_E(C(c,1),c) \geq F_E(C(c,2),c) \geq F_E(C,c)$ for all C in $\mathcal{C}_2$. The correctness of the claim for any type-2 cell c of $\mathcal{T}'$ then follows.

Finally we need to show that in **Algorithm 6**, $F_E(C(c,j),c) \geq F_E(C,c)$ actually holds for $j = 1, 2$ and all $C \in \mathcal{C}_j$. If c_j is a type-2 cell, then $F_E(C,c) = F_E(C,c_j)$ for all $C \in \mathcal{C}_j$. In **Algorithm 6**, we set $C(c,j)$ to be $C(c_j)$. Therefore $F_E(C(c,j),c_j) = val(c_j)$ by induction hypothesis. Again by induction hypothesis, we have $F_E(C(c,j),c_j) \geq F_E(C,c_j)$ for any $C \in \mathcal{C}_j$. Thus, we get $F_E(C(c,j),c) = F_E(C(c,j),c_j) \geq F_E(C,c_j) = F_E(C,c)$. If c_j is a type-1 cell, let C be a cluster in $\mathcal{C}_j$ other than $C(c_j)$. By the previous definition, we know that $F_E(C,c) \leq F_E(C(c_j),c) = F_E(C(c,j),c)$ must hold (Note that $C(c,j)$ is set to be $C(c_j)$). This completes the proof. □

5.2. Analysis of Complexity

In this subsection, we analyze the time and space complexities of **Algorithm 5**. We first bound the running time and space of AssistedAIDecomposition (**Algorithm 4**).

Definition 3. During the execution of AssistedAIDecomposition, a distance-node v is said to be **refered to** by a box-node u, if v appears in the list L when AssistedDecomposition(**Algorithm 1**) is called on u.

The following lemma is a straightforward observation of the algorithm AssistedAIDecomposition.

Lemma 4. *The running time of AssistedAIDecomposition, excluding the time to build the distance-tree, is* $\sum_v O(R(v) + H(v) + I(v))$*, where the sum is over all distance-nodes v generated by AssistedAIDecomposition, $R(v)$ is the number of times v is referred to, $H(v)$ is the number of times v appears as a parameter of HandleType1Cell(***Algorithm 2***), and $I(v)$ is the number of times v has ever appeared in procedure SearchTail(***Algorithm 3***) as u'. The size of the box-tree built by AssistedAIDecomposition is* $\sum_v O(R(v) + H(v))$.

From now on, we use $R(v), H(v), I(v)$ as described in the above lemma. The following lemma gives a bound on $R(v)$. This is a result from applying the Packing Lemma in [5]. The proof is in **Section 6.2**.

Lemma 5. *For any distance-node v generated by AssistedAIDecomposition(P, $\beta, \mathcal{T}_a$), $R(v) = O(\log|P|)$.*

The following lemma gives a bound on $H(v)$.

Lemma 6. *A call to AssistedDecomposition where v appears in the list L can only generate a constant number of subsequent calls to HandleType1Cell with v as a parameter from step 4.1.2. As a consequence, $H(v) = O(R(v))$.*

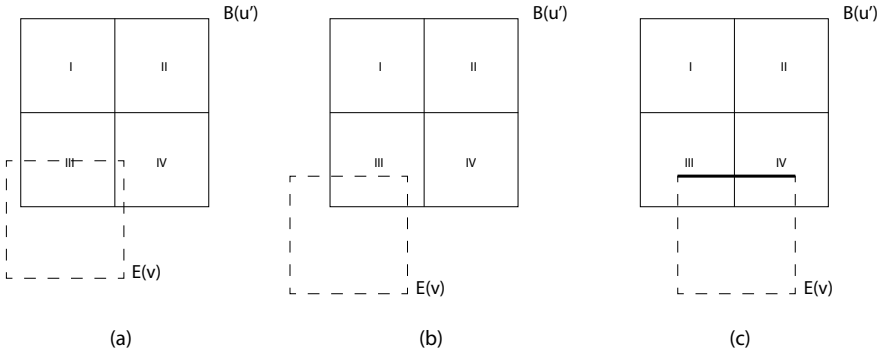

Figure 4: An example illustrating location of box $B(u')$ (Divided equally into regions I to IV by lines in it) and $E(v)$. If Step 4.1.3 is executed, then the configuration should be like (c): one of the edge (the bold edge) of $E(v)$ is completely in $B(u')$ but cut by a line in $B(u')$. (a) or (b) shows what happens if no whole edge is cut. (a): The size of $E(v)$ exceeds half size of $B(u')$. (b): only one of regions I-IV intersects $E(v)$. Either of them prevents step 4.1.3 from execution.

Proof. Assume that a call to AssistedDecomposition performs step 4.1.2 so that a call to Handle-Type1Cell$(u, \beta, v, T_p, r_c, \mathcal{T}_a)$ is made. Note that the execution of HandleType1Cell can have three possible outcomes. The first is that a type-1 cell is generated and the subroutine returns. The second is that HandleType1Cell invokes AssistedDecomposition, with L set to have v_1 and v_2, the two children of v, which means that later calls to AssistedDecomposition cannot have v appear in L, and thus v cannot be a parameter of later calls to HandleType1Cell. The third is that **at least 2**, and **at most** 2^d calls to HandleType1Cell are made. Therefore, if we want to bound the number h of calls to HandleType1Cell that stem from the execution of HandleType1-Cell$(u, \beta, v, T_p, r_c, \mathcal{T}_a)$ with v as a parameter, we can simply count the number h' of calls to HandleType1Cell which have the third type of outcome. Clearly $h = O(h')$. We will prove that $h' = O(1)$, from which the lemma follows.

We start with a few definitions. Let z_i denote the edge length of $E(v)$ along the i-th coordinate axis. Consider a call to HandleType1Cell$(u', \beta, v, T_p, r_c, \mathcal{T}_a)$ (which stems from the call to HandleType1Cell$(u,\ \beta, v, T_p, r_c, \mathcal{T}_a)$; for convenience, we always assume that it is the case for any call to HandleType1Cell in this proof). Let $W(u') \subseteq \{1, 2, \ldots d\}$ denote the set of integers i such that there exist two points, q_1 and q_2 in $B(u') \cap E(v)$ with $|x_i(q_1) - x_i(q_2)| = z_i$, where $x_i(q)$ denotes the i-th coordinate of a point q.

Note that if HandleType1Cell($u', \beta, v, T_p, r_c, \mathcal{T}_a$) invokes HandleType1Cell, for any of its calls to HandleType1Cell($u'', \beta, v, T_p, r_c, \mathcal{T}_a$), we have $W(u'') \subsetneq W(u')$. This is because if step 4.1.3 of Handle-Type1Cell($u', \beta, v, T_p, r_c, \mathcal{T}_a$) is executed, the length of any edge of $B(u') \cap E(v)$ should be smaller than $\frac{size(B(u'))}{2}$. It is clear from the algorithm that $B(u'')$ is one of the 2^d boxes equally decomposing $B(u')$ (or $B(u_t)$), and $B(u'') \cap E(v) \neq \emptyset$. Note that there must be another call to HandleType1Cell($u_1'', \beta, v, T_p, r_c, \mathcal{T}_a$) with $B(u'')$ being another one of the 2^d boxes equally decomposing $B(u')$ and $B(u_1'') \cap E(v) \neq \emptyset$. All of the above can happen only if one dimension of $E(v)$ is split with $B(u')$ decomposed equally into 2^d smaller boxes, making $W(u'') \subsetneq W(u')$. See Figure 4 for easy understanding.

Let $S(i)$ denote the maximum number of subsequence calls to HandleType1-Cell that stem from the execution of HandleType1Cell($u', \beta, v, T_p, r_c, \mathcal{T}_a$) with $|W(u')| = i$. We first show that $S(1) \leq 2^d$. If $|W(u')| = 1$, then for any call to HandleType1 Cell($u'', \beta, v, T_p, r_c, \mathcal{T}_a$) made by HandleType1Cell(u', β, v, $T_p, r_c, \mathcal{T}_a$), we have $|W(u'')| = 0$ and as a result, HandleType1Cell(u', β, v, T_p, r_c, $\mathcal{T}_a$) will not call HandleType1Cell. Since HandleType1Cell ($u', \beta, v, T_p, r_c, \mathcal{T}_a$) makes at most 2^d calls to HandleType1Cell, we get $S(1) \leq 2^d$. For HandleType1-Cell($u', \beta, v, T_p, r_c, \mathcal{T}_a$) with $|W(u')| = i + 1$, it makes at most 2^d calls to HandleType1Cell($u', \beta, v, T_p, r_c, \mathcal{T}_a$), which make no more than $S(W(u'')) \leq S(i)$ descendant calls to HandleType1Cell. Thus, $S(i + 1) \leq 2^d S(i)$. This means that $S(i) \leq (2^d)^i$. Note by definition we have $W(u) \subset \{1, 2, \ldots d\}$ and $|W(u)| \leq d$. Therefore, $h' \leq S(d) \leq (2^d)^d$, which completes the proof. □

Due to the fact that the whole procedure of AssistedAIDecomposition is no more than a traversal of $\mathcal{T}_a$, we have the following observation.

Observation 2. $\sum_v I(v) = O(sizeof(\mathcal{T}_a))$.

Then the running time of AssistedAIDecomposition is given by the following theorem.

Theorem 3. *AssistedAIDecomposition*$(P, \beta, \mathcal{T}_a)$ *generates a box-tree of size* $O(|P| \log |P|)$ *within time* $O(|P| \log |P| + sizeof(\mathcal{T}_a))$.

We can now bound the running time of the divide-and-conquer approach.

Theorem 4. *For a given set* $\mathcal{C}$ *of clusters, and any influence function* $F(C, q)$ *satisfying the properties (P.1) to (P.5), if a data structure described in (P.5) is available for* $\mathcal{C}$*, then BuildAIVD*($\mathcal{C}, \epsilon$,*NULL) builds a* $(1 - \epsilon)$*-approximate IVD for* $\mathcal{C}$ *within* $O(T_2(N)N \log^2 N)$ *time, where* N *is the size of the input* $\mathcal{C}$ *and* $\epsilon > 0$ *is a sufficiently small constant.*

Proof. To prove this theorem, we first consider the running time of Build-AIVD($\mathcal{C}', \epsilon, \mathcal{T}_a'$) on a subset $\mathcal{C}'$ of clusters. Obviously, BuildAIVD($\mathcal{C}', \epsilon, \mathcal{T}_a'$) is a descendant of BuildAIVD($\mathcal{C}, \epsilon$,NULL) in the recursion tree of BuildAIVD. Let $N' = \sum_{C \in \mathcal{C}'} |C|$ and M_T be the size of $\mathcal{T}_a'$ (0 if $\mathcal{T}_a'$ is NULL). Then step

1 takes $O(N' \log N') + O(M_T)$ time to build a box-tree $\mathcal{T}'$, by the previous analysis on the running time of **Algorithm 4**. The running time of step 2 is $O(T_2(N) \cdot sizeof(\mathcal{T}'))$, which has been shown to be $O(T_2(N))O(N' \log N')$ by **Theorem 3**. Step 3, if executed, takes $O(T_2(N))O(N' \log N')$. Step 4, if executed, makes two calls to BuildAIVD, one for each of the subsets $\mathcal{C}_1$ and $\mathcal{C}_2$ of $\mathcal{C}$. Two box-trees are generated with sizes clearly no larger than $O(N' \log N')$, which is the size of $\mathcal{T}'$. The merging process of the two (**Algorithm 6**) takes $O(T_2(N))O(N' \log N')$ time. Thus, the total running time is $O(T_2(N))$ $O(N' \log N') + O(M_T)$, plus the running time of the two possible recursive calls on $\mathcal{C}_1$ and $\mathcal{C}_2$.

The term $O(M_T)$ is seemingly difficult to analyze, as it does not depend on N'. To get around this obstacle, we consider the parent BuildAIVD$(\mathcal{C}'', \epsilon, \mathcal{T}_a'')$ of BuildAIVD$(\mathcal{C}', \epsilon, \mathcal{T}_a')$ in the recursion tree of BuildAIVD. Since $M_T = O(N''$ $\log N'')$, where $N'' = \sum_{C \in \mathcal{C}''} |C|$, and the running time of BuildAIVD$(\mathcal{C}'', \epsilon, \mathcal{T}_a'')$ already has an term $O(N'' \log N'')$, this means that we can charge part of the cost (*i.e.,* $O(M_T)$) of BuildAIVD$(\mathcal{C}', \epsilon, \mathcal{T}_a')$ to its parent BuildAIVD$(\mathcal{C}'', \epsilon, \mathcal{T}_a'')$ in the recursion tree and let its parent absorb the additional term of $O(N'' \log N'')$.

To derive the recurrence formula of the running time of BuildAIVD on $\mathcal{C}'$, we need to know the sizes $\sum_{C \in \mathcal{C}_i} |C|, i = 1, 2$, of the two subsets $\mathcal{C}_1$ and $\mathcal{C}_2$ of clusters. There are two possibilities: (1) Both subsets have size no more than $\frac{5}{6}$ of the size of $\mathcal{C}'$, *i.e.,* $\sum_{C \in \mathcal{C}_i} |C| \leq \frac{5}{6} \sum_{C \in \mathcal{C}'} |C|$, $i = 1, 2$, and (2) one of the subsets, say $\mathcal{C}_2$, has size larger than $\frac{5}{6}$ of the size of $\mathcal{C}'$. For case (1), the recurrence formula can be written as

$$T(N') = T(\frac{5}{6}N') + T(\frac{1}{6}N') + O(T_2(N))O(N' \log N').$$

For case (2), let $N_2 = \sum_{C \in \mathcal{C}_2} |C| > \frac{5}{6} \sum_{C \in \mathcal{C}'} |C|$. Consider step 4 of **Algorithm 5**. Since $\sum_{C \in \mathcal{C}_1} |C| < \frac{1}{6} \sum_{C \in \mathcal{C}'} |C|$, C_m, found in step 4.2, should satisfy inequality $|C_m| > \frac{1}{3} \sum_{C \in \mathcal{C}'} |C|$. Since $C_m \leq C_{m+1} \leq C_{m+2} \leq \ldots$, $\mathcal{C}_2$ has no more than three clusters, which means that the execution of **Algorithm 5** on it will go to step 3. The running time will then be $O(T_2(N))O(N_2 \log N_2)$ (with a term absorbed by BuildAIVD$(\mathcal{C}', \epsilon, \mathcal{T}_a')$ as discussed above), which is $O(T_2(N))O(N' \log N')$ since $N' \geq N_2$. The recurrence formula then can be written as,

$$T(N') = T(\frac{1}{6}N') + O(T_2(N))O(N' \log N'),$$

which is an asymptoticly looser bound than

$$T(N') = T(\frac{5}{6}N') + T(\frac{1}{6}N') + O(T_2(N))O(N' \log N').$$

To summarize the running time of BuildAIVD$(\mathcal{C}', \epsilon, \mathcal{T}_a')$, with M_T absorbed by its parent, will depend on N'. The recurrence formula can be written as

$$T(N') = T(\frac{5}{6}N') + T(\frac{1}{6}N') + O(T_2(N))O(N' \log N').$$

The Theorem follows by solving the formula. □

6. Appendix

In this appendix, we give the details of AI decomposition and the proofs of some lemmas. They are necessary for the understanding of the proposed algorithms. Some of them are either directly from [5] or modified from similar lemmas/theorems in [5]. We include them in this appendix for self completeness. Throughout the appendix, let $n = |P|$.

6.1. More Detailed Description of AI Decomposition

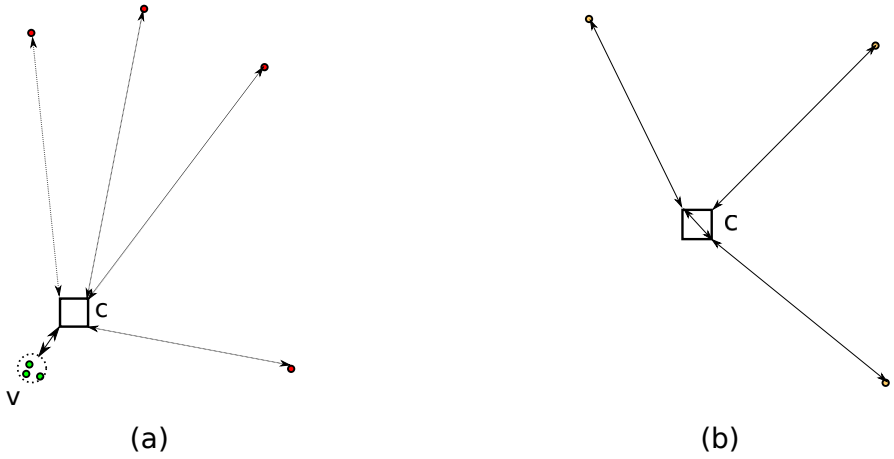

Figure 5: An example of type-1 cell(a) and type-2 cell(b). In (a) the cell c is very close to a point set v compared to other input points. In (b) the diameter of c is small compared to its distance to input points.

In this section we provide a more detailed introduction to AI decomposition. The basic idea of AI Decomposition to build a partition induced by the input point set P is recursion: it starts the procedure with a large enough bounding box of P, which is also the root of the box-tree, decomposes the large box into smaller subregions, makes the smaller subregions the children of the large box and recursively continues the decomposition on the smaller subregions. There are two possibility that a box B is decomposed into smaller regions during the above described procedure. First, B can be equally decomposed into 2^d smaller boxes. Second, through a process to be discussed later, a smaller sub-box B' within B is computed, and then B is decomposed into B' and $B \setminus B'$. The decomposition process ends for a box if if it meets the criteria for type-1 or type-2 cell, *i.e.,* if it is very close to a subset of input points (in P) or it is small enough compared to its distance to points in P (See Figure 5 for an illustration of the two types of cells).

The crucial part of the above method is to efficiently verify the stopping condition for any box B. Information about distance between B and points in

P is necessary for such verification. However, it is very inefficient to compute the distance between every point in P and B every time a box B is considered. To ensure efficiency, multiple input points in P are viewed as a single "heavy" point, thus the number of points involved in distance computations with B is reduced. This idea is realized by a pre-computed data structure called the *distance-tree*. The distance-tree is a binary tree where each node (called a *distance-node*) v is associated with a subset $P(v)$ of P (See Figure 6 for an illustration of a distance tree). Under certain conditions, all points in $P(v)$ can be viewed as a single "heavy" point. During the recursion, a list L of distance nodes (or a list of "heavy" points) is maintained. To determine whether B is a type-1 or type-2 cell, or B needs further decomposition, only distances between B and the "heavy" points in L are checked, instead of considering the distances to all points in P.

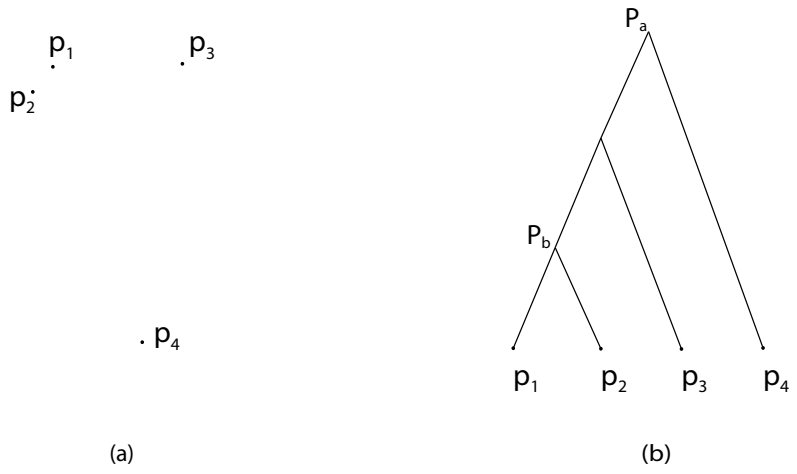

Figure 6: An example of distance-tree (b) built for 4 points (a). There is a leaf in the distance-tree for every input point. Every node represents a point set. For example every leaf represents a single point set. P_b in (b) represents set $\{p_1, p_2\}$, and the root P_a represents the whole input point set $\{p_1, p_2, p_3, p_4\}$.

Below are the algorithms for building the distance-tree (the preprocessing step) and the AI decomposition, where $0 < \beta < 1/2$ and polynomially bounded function $\mathcal{P}(\cdot)$ (in step 4) are controlling parameters which depend on individual problem and the desired accuracy. The most important part is the Decomposition procedure, which is used to determine whether a box-tree node is a type-1 or type-2 cell, or it needs further decomposition.

Algorithm 7 Preprocessing(P, β)

Input: A set P of n points in $\mathbb{R}^d$, and an error tolerance $0 < \beta < 1/2$.
Output: A tree structure T_p, in which every node v stores a value $s(v)$, an input point $l(v)$, and is associated with a bounding box $E(v)$ in $\mathbb{R}^d$.

1: Compute a 12-well separated pair decomposition [3] $W = \{(A_1, B_1), (A_2, B_2), \ldots, (A_m, B_m)\}$ of P.
2: Construct a graph $G(W)$ by connecting the representatives of A_i and B_i, for every $(A_i, B_i) \in W$.
3: Build a min-priority queue Q for all edges in $G(W)$, base on their edge lengths.
4: Build a tree T_p in the following bottom-up manner.

For each $p \in P$, there is a leaf node v_p in T_p (*i.e.*, T_p is initially a forest of $|P|$ single-node trees), with $s(v_p) = 0$, $l(v_p) = p$, and $E(v_p)$ and $E'(v_p)$ both being 0-sized bounding boxes containing p.

While T_p is not a single tree **Do**

- Extract from Q the shortest edge $e = (p_1, p_2)$ with edge length $w(e)$. If v_{p_1} and v_{p_2} are leaves of two different trees in T_p rooted at v_1 and v_2, then create a new node v in T_p as the parent of v_1 and v_2, and let $s(v) = s(v_1) + s(v_2) + w(e)$, $l(v)$ be either $l(v_1)$ or $l(v_2)$, $E'(v)$ be the box centered at $l(v)$ and with size $\frac{4 \cdot s(v)}{\beta}$, and $E(v)$ be the box centered at $l(v)$ and with size $\frac{8 \cdot s(v)}{\beta}$.

Algorithm 8 Decomposition(u, β, L, T_p, r_c)

Input: A box-node u with box $B(u)$, error tolerance $\beta > 0$, distance-tree T_p, linked list L, and a value r_c. **Output:** A subtree of T_q rooted at u.

1: **While** $\exists$ v in L such that the length of at least one edge of $B(u) \cap E(v)$ is no smaller than $\frac{size(B(u))}{2}$ **do**

- Replace v in L by its two children in T_p, if any.

2: Let $D(u)$ be the diameter of $B(u)$. For each node v in L **do**

2.1 Let r_{min} be the distance between $B(u)$ and $l(v)$.

2.2 If $D(u) < r_{min}\beta/2$, remove v from L, and if $r_c > r_{min}$, let $r_c = r_{min}$.

3: If L is empty, return, and $B(u)$ becomes a **type-2** cell.

4: If there is only one element v in L, let r_{min} be the smallest distance between $l(v)$ and $B(u)$.

4.1 If $\frac{r_{min}+D(u)}{r_c} < \frac{\beta}{2\mathcal{P}(n)}$,

4.1.1 If $E(v) \cap B(u) = \emptyset$ or v is a leaf node in T_p, $B(u)$ is a **type-1** cell dominated by v. Return.

4.1.2 Let B' be the smallest hypercube in $B(u)$ fully containing $B(u) \cap E(v)$. Create two box nodes u_0 and u_1, with u_0 corresponding to B' and u_1 corresponding to the difference of $B(u)$ and B'. Let u_0 and u_1 be children of u in T_q. In this case, u_1 is a **type-1** cell dominated by v.

4.1.3 Replace v in L by its two children v_1 and v_2 in T_p. Call Decomposition$(u_0, \beta, L, T_p, r_c)$, and return.

5: Decompose $B(u)$ into 2^d smaller boxes, and make the corresponding nodes $u_1, u_2, \ldots, u_{2^d}$ as the children of u in T_q. Call Decomposition$(u_i, \beta, L, T_p, r_c)$ for each u_i. Return.

Algorithm 9 AI-Decomposition(P, β)

Input: A set P of n points in $\mathbb{R}^d$, and a small error tolerance $\beta > 0$.
Output: A box-tree T_q.

1: Run the preprocessing algorithm on P and obtain a distance-tree T_p. Let u be the root of T_p. View $E(u)$ as a box-tree node. Run Decomposition($E(u), \beta, \{u\}, T_p, \infty$).
2: Output the box-tree rooted at $E(u)$ as T_q.

6.2. Proof of Lemma 5

In this section we will prove Lemma 5. These proofs are directly from the full paper of [5] with some minor modifications, since many facts about AI Decomposition were already discussed in [5]. Nonetheless, for completeness we include them here because slight changes to these facts and proofs are required for the correctness of our approach.

*6.2.1. Lemmas Necessary for Proving **Lemma** 5*

Lemma 7. *Let u and u_p be two nodes in the box-tree $\mathcal{T}$ generated by **Algorithm 4** such that u_p is the parent of u, and u is not a type-1 cell. Then $B(u)$ is at most half the size of $B(u_p)$. Furthermore, if u_p has at least two non-type-1-cell children, then $B(u)$ is one of the boxes obtained by decomposing $B(u_p)$ equally into 2^d smaller boxes.*

Proof. We prove this by induction on the recursive process of **Algorithm 5**. For the basis, at the top level of the recursion, **Algorithm 9** is used to build a box-tree. A node u is generated when **Algorithm 8** is processing its parent u_p, by either decomposing $B(v_p)$ equally using quad-tree decomposition, or in step 4, by using a minimum sub-box of $B(v_p)$ to cover $B(v_p) \cap E(v)$ for some distance node v. In the former case, the size of $B(v)$ is exactly half the size of $B(u_p)$. In the latter case, the size of $B(v)$ cannot exceed the maximum edge length x of $B(v_p) \cap E(v)$. Note that x cannot exceed half the edge length of $B(v_p)$, since otherwise v would have been removed from L in step 1, a contradiction. In either case, $B(u)$ is at most half the size of $B(u_p)$. The only possibility that u_p have two non-type-1-cell children is that $B(u_p)$ is decomposed into 2^d smaller boxes by quad-tree decomposition. Thus $B(u)$ can only be one of the smaller boxes.

The induction step is trivial. Say $\mathcal{T}$ is produced by **Algorithm 4** assisted by $\mathcal{T}_a$, and the lemma holds for $\mathcal{T}_a$, Then the lemma automatically holds for $\mathcal{T}$, since $\mathcal{T}$ is a pruned version of $\mathcal{T}_a$. □

The following lemma shows a property of the distance-tree T_p that will be used in the proof of **Lemma 5**.

Lemma 8. *Let v be any node in T_p other than the root, and r be the minimum distance between any input point in $P(v)$ and any input point in $P \setminus P_v$. Let v_p be v's parent, then $s(v_p) \leq 2nr$.*

PROOF. Let r_G be the minimum length of any edge in $G(W)$ connecting some input point in $P(v)$ to some input point in $P \setminus P(v)$. Since $G(W)$ is a 2-spanner for P, $r_G \leq 2r$. By **Algorithm 7**, we know that v_p (and $P(v_p)$) is formed by a sequence of no more than n merge operations on the nodes of T_p. The last these operations extract an edge connecting some input point in $P(v)$ to some input point in $P \setminus P(v)$, whose length is no larger than r_G. Each merge operation contributes to $s(v_p)$ a value no bigger than r_G, since the edge e extracted from Q by **Algorithm 7** has a length $w(e)$ no larger than r_G. Hence, $s(v_p) \leq 2r_G \leq 2nr$. □

The following packing lemma has been proved in [5] and is the key to prove **Lemma 5**.

Lemma 9 (Packing Lemma [5]). *Let o_c be any point in $\mathbb{R}^d$, and S_{in} and S_{out} be two d-dimensional boxes (*i.e., *axis-aligned hypercubes) co-centered at o_c and with edge lengths $2r_{in}$ and $2r_{out}$, respectively, with $0 < r_{in} < r_{out}$. Let $\mathcal{B}$ be a set of mutually disjoint d-dimensional boxes such that for any $B \in \mathcal{B}$, B intersects the region $S' = S_{out} - S_{in}$ (i.e., the region sandwiched by S_{in} and S_{out}) and its edge length $L(B) \geq C \cdot r$, where r is the minimum distance between B and o_c and C is a positive constant. Then $|\mathcal{B}| \leq C'(C,d)\log(r_{out}/r_{in})$, where $C'(C,d)$ is a constant depending only on C and d.*

*6.2.2. Proof of **Lemma 5***

PROOF. For simplicity of our argument, we will make some small modifications to **Algorithm 1**.

If a node u_a generated in step 5.1, all distance node in L at step 5.1 is considered referred to by u_a. (Although we do not run **Algorithm 1** recursively on u_a).

If in step 5 only one child u_b of u is generated, we define $B_m(u_b)$ to be either the a box which wholly contains $B(u_b)$, and is wholly contained in $B(u)$, and whose size is 1/4 the size of $B(u)$, or simply $B(u_b)$ if the size of $B(u_b)$ is no smaller than 1/4 the size of $B(u)$. We then also generate a node u'_b which becomes a child of u. $B_m(u'_b)$ and $B(u'_b)$ is set to be a box wholly contained in $B(u)$, and $B_m(u'_b)$ is disjoint with $B_m(u_b)$, and the size $B_m(u'_b)$ is no smaller than 1/4 the size of $B(u)$. Note such a $B_m(u'_b)$ exists since by **Lemma 7** and the definition of $B_m(u_b)$, the size of $B_m(u_b)$ is no smaller than half the size of $B(u)$. All distance nodes in L at step 5 is considered referred to by u_b and u'_b.

Note that the modification will only increase the value of $R(v)$, therefore if we are able to prove the bound on $R(v)$ for the modified algorithm, the lemma then follows. From now on the modified version of **Algorithm 1** is assumed.

We first consider the case that v is the root of T_p. In this case, v is referred to only once, by the root of the box-tree $\mathcal{T}$, and the statement is trivially true. Thus, we assume that v is an arbitrary distance-node other than the root of T_p.

We now observe that if a box-node u refers to v, then either all or none of u's children refers to v (the latter case happens if v is removed from L when u is the current box-node, or HandleType1Cell is called and as a result, v in L

will be replaced by its two children if later calls to AssistedDecomposition are made). This means that we only need to count those box-nodes u which refer to v and have v remove from L when u is the current box-node. The reason is that although we do not count those box-nodes, say u', which do not remove v from their L lists when they become the current box-nodes, the number of box-nodes (*i.e.*, the 2^d children of u') which refer to v at the next level of recursion increases exponentially. This implies that the total number of box-nodes which refer to v but are not counted is no more than the total number of box-nodes which are counted. Thus, we can safely ignore those u'. Let U_v denote the set of u's which are counted.

We define a mapping Φ on U_v. Let u' be the parent of u in $\mathcal{T}$ (if existing). We first define $u_h(u)$ for u generated in **Algorithm 2**: we assume that **Algorithm 2** is called by step 4.1.2 of **Algorithm 1** when processing $u_h(u)$. $\Phi(u)$ is defined as

$$\Phi(u) = \begin{cases} B(u_h(u)) & \text{if } u \text{ is generated in } \textbf{Algorithm 2}, \\ B_m(u) & \text{if } B_m(u) \text{ is defined(Recall how we modified } \textbf{Algorithm 1}), \\ B(u) & \text{otherwise.} \end{cases}$$

It is not hard to see that for $u_1, u_2 \in U_v$ and $u_1 \neq u_2$, $\Phi(u_1)$ and $\Phi(u_2)$ are either disjoint, or they are equal. The latter case happens when a HandleType1Cell generates multiple calls to AssistedDecomposition. By **Lemma 6**, there are at most constant number of u that share the same Φ. Let $\mathcal{B} = \{\Phi(u) \mid u \in U_v\}$. It is sufficient to show $|\mathcal{B}| = O(\log n)$. Our strategy is to use **Lemma 9** for counting. To do this, we prove that there exist boxes B_{out} and B_{in} with sizes s_{out} and s_{in} respectively and a constant c_0 depending only on d and β such that all of the following hold:

1. B_{out} and B_{in} are co-centered at $l(v)$.
2. Every box in $\mathcal{B}$ intersects B_{out}.
3. No box in $\mathcal{B}$ is contained entirely in B_{in}.
4. $\frac{s_{out}}{s_{in}}$ is bounded by some polynomial of n.
5. For any $B \in \mathcal{B}$, $s \geq c_0 r$, where r is the shortest distance between B and $l(v)$, s is the size of B, and c_0 is some positive constant depending on d and β.

Clearly, if all of the above hold, then by **Lemma 9**, we have $|\mathcal{B}| = O(\log n)$.

Let r' be the minimum distance between a point in $P(v)$ and a point in $P \setminus P(v)$.

We first determine B_{out}. Let v' be the parent of v in T_p. Let s' be the size of $E(v')$. We choose $s_{out} = 7s'$ and claim that for every box-node u that refers to v, $B(u)$ fully contained inside B_{out}. Let u' be the ancestor of u such that v' is removed from L in Step 1 of **Algorithm 1** when processing u' , or u' is processed by **Algorithm 1** called in step 2 of **Algorithm 2** (where v' is also removed from L). Note that u' must exist since these are the only two ways for v to appear in L. If v' is removed from L in Step 1 of **Algorithm 1**, we know that $B(u')$ intersects $E(v')$ and has at most twice the size of $E(v')$. Therefore,

$B(u')$ is entirely contained inside B'_{out}, where B'_{out} is the box centered at $l(v')$ and with a size $5s'$. Thus $B(u) \subseteq B(u') \subseteq B'_{out}$. If v' is removed from L in step 2 of **Algorithm 2**, note the length of at least one edge of $B(u') \cap E(v')$ should be no smaller than half the size of $B(u')$, again $B(u')$ intersects $E(v')$ and has at most twice the size of $E(v')$, thus $B(u')$ is entirely contained inside B'_{out}. Therefore $B(u) \subseteq B(u')B'_{out}$. Thus, in either case, $B(u)$ is contained inside B'_{out}. Since $\|l(v) - l(v')\| \leq s(v') \leq \beta s'/8 \leq s'$, B'_{out} is completely inside B_{out}. Thus, the above claim is true.

From this claim, it is clear that every box in $\mathcal{B}$ intersects B_{out}, whose size is $s_{out} = 7s' = \frac{56s(v)}{\beta} \leq \frac{112n}{\beta} r'$.

Let $\beta_0 = \frac{2(1+\beta)\mathcal{P}(n)}{\beta}$. We choose $s_{in} = \frac{r'}{14\sqrt{d}(1+\beta_0)}$, and claim that for every u that refers to v, $\Phi(u)$ cannot be completely inside B_{in}. Suppose this is not the case, and there exists such a box-node u whose $\Phi(u)$ is fully contained inside B_{in}.

First of all, it is easy to see that such a box-node u cannot be the root of the box-tree $\mathcal{T}$, since otherwise, $B(u)$ should be contained inside B_{in}. But this cannot be the case, as $B(u)$ contains all input points and its size is obviously larger than that of B_{in}. (Note that in this case, $\Phi(u) = B(u)$.)

Next, we show that such a u (i.e., whose $\Phi(u)$ is inside B_{in}) is not generated in **Algorithm 2**, which stems from step 4.1.2 of **Algorithm 1** when processing u'. Suppose, for contradiction, u is generated in **Algorithm 2**. Let v' be the parent of v in T_p. Then $E(v')$ does not fully contain $B(u')$, since otherwise it would have been deleted from L in Step 1 of **Algorithm 1**, instead of in **Algorithm 2**, when processing u'. Note that since v' contains at least one input point that is not in $P(v)$, the diameter of $E(v')$ must be greater than r'. This means that $E(v')$ is at least 6 times larger than B_{in} in size. The distance between $l(v)$ and $l(v')$ (*i.e.,* the centers of B_{in} and $E(v')$, respectively) satisfies the inequalities $\|l(v) - l(v')\| \leq s(v') \leq R\frac{\beta}{8} \leq \frac{R}{16}$, where R is the size of $E(v')$. This means that B_{in} is fully contained in $E(v')$, and therefore cannot contain $B(u')$, which is $\Phi(u)$. This is a contradiction, and thus u cannot be generated in Step 4.

Finally, we show that u cannot be generated in Step 5 of **Algorithm 1**. Suppose u is generated in Step 5 when processing u', where u' is the parent of u in $\mathcal{T}$. Since $\Phi(u)$ is contained in B_{in} (by assumption), we know that $B(u')$, which has at most 4 times the size of $\Phi(u)$(Recall how we modified the algorithm in the beginning of the proof), must be contained in a box B'_{in} centered at $l(v)$ and with a size $\frac{r'}{2\sqrt{d}(1+\beta_0)}$. This means $D(u') \leq \frac{r'}{2(1+\beta_0)}$, where $D(u')$ is the diameter of $B(u')$. Let r'' be the distance between $l(v)$ and $B(u')$. Then, by the fact that B'_{in} contains $B(u')$, we have $r'' \leq \frac{r'}{2\sqrt{d}(1+\beta_0)}$. Combining the above two inequalities, we get $r'' + D(u') \leq \frac{r'}{(1+\beta_0)}$. For any point $p \in P \setminus P(v)$, let r_p be the distance between p and $B(u')$, and q' be the closest point on $B(u')$ to p. Then by the triangle inequality, we know that the distance $\|l(v) - q'\|$ between $l(v)$ and q' is no larger than $r''+D(u')$. Thus, we have $\|l(v)-q'\| \leq \frac{r'}{(1+\beta_0)}$. Also, by the definition of r', we know that the distance $\|p - l(v)\|$ between p and $l(v)$

is no smaller than r'. By the triangle inequality (in the triangle $\Delta l(v)pq'$), we know $r_p = \|p - q'\| \geq \|p - l(v)\| - \|l(v) - q'\| \geq r' - \frac{r'}{(1+\beta_0)} = \frac{\beta_0 r'}{(1+\beta_0)}$. Therefore, we have $\frac{r''+D(u')}{r_p} \leq \frac{1}{\beta_0} = \frac{\beta}{2(1+\beta)\mathcal{P}(n)}$. This implies $\frac{D(u')}{r_p} \leq \frac{1}{\beta_0} \leq \frac{\beta}{2}$. Since the above inequality holds for every point in $P \setminus P(v)$, this indicates that every such point must be recorded for u' (see Step 2 of **Algorithm 1**). By **Algorithm 1**, we know that r_c stores the minimum recorded distance. Also, note that a point in P is recorded for u' if and only if it is in $P \setminus P(v)$. Therefore, some $p \in P \setminus P(v)$ gives rise to the recorded distance r_c. By **Lemma 1**, we know $r_p \leq (1+\beta)r_c$. Thus, we have $\frac{r''+D(u')}{r_c} \leq \frac{\beta}{2\mathcal{P}(n)}$. Since each point $p \in P \setminus P(v)$ is recorded for u' and v is referred to by u (which is a child of u'), it must be the case that after finishing Step 2 of **Algorithm 1** in the recursion for u', v is the only distance-node in L. Then, by the fact of $\frac{r''+D(u')}{r_c} \leq \frac{\beta}{2\mathcal{P}(n)}$, we know that u' will be processed in Step 4, which includes the generation of node u, instead of Step 5. This is a contradiction.

Summarizing the above three cases, we know that every box in $\mathcal{B}$ is not fully contained in B_{in}.

From the above discussion, we know that the sizes of B_{out} and B_{in} satisfy the following inequality

$$\frac{s_{out}}{s_{in}} \leq \frac{1568\sqrt{d}n(1 + \frac{2(1+\beta)\mathcal{P}(n)}{\beta})}{\beta}.$$

This means that the ratio of $\frac{s_{out}}{s_{in}}$ is bounded by a polynomial of n.

The only remaining issue is to show that for any $u \in U_v$, the size s of $\Phi(u)$ and the distance r between $\Phi(u)$ and $l(v)$ satisfy the relation of $s \geq c_0 r$ for some constant c_0. Note that such a relation is trivially true for any c_0 if u is the root of $\mathcal{T}$, since in this case $B(u)$ contains all input points and the distance r is 0. Thus its distance to $l(v)$ is 0. Hence, we assume that u is not the root of $\mathcal{T}$.

For any box-node $u_0 \in \mathcal{T}$ and any distance node $v_0 \in T_p$, let $r(u_0, v_0)$ be the shortest distance between $B(u_0)$ and $l(v_0)$. We consider two cases.

1. u is generated in **Algorithm 2** which stem from step 4.1.2 of **Algorithm 1** when processing u'. In this case, $\Phi(u) = B(u')$. Let v' be the parent of v in T_p. We consider two sub-cases, depending on whether $E'(v')$ intersects $B(u')$ (see **Algorithm 7** for the definition of $E'(v')$).
   (a) $E'(v')$ intersects $B(u')$. In this case, since v' is not removed from L in Step 1 of **Algorithm 1** when processing u', some part of $B(u')$ must be outsides $E(v')$. (This is because $E'(v')$ is co-centered at $l(v')$ with $E(v')$ and is of half the size of $E(v')$. If $B(u')$ is fully inside $E(v')$, then an edge length of $B(u') \cap E(v')$ will be larger than half the size of $B(u')$, and therefore v' will be removed from L in Step 1.) This means that the size of $B(u')$ is at least half the size of $E'(v')$, which is $\frac{2s(v')}{\beta}$. Thus, the diameter $D(u')$ of $B(u')$ exceeds $\frac{2\sqrt{d}s(v')}{\beta}$. Furthermore, since $E'(v')$ intersects $B(u')$, we have $r(u', v') \leq \frac{2\sqrt{d}s(v')}{\beta}$ (by the

definition of $r(u', v')$ and the size of $E'(v')$). Also since $P(v')$ contains both $l(v)$ and $l(v')$, the distance between $l(v)$ and $l(v')$ is upper-bounded by the diameter $s(v')$ of $P(v')$, i.e., $\|l(v) - l(v')\| \leq s(v')$. Thus, we have $r(u', v) \leq \|l(v) - l(v')\| + r(u', v') \leq s(v') + \frac{2\sqrt{d}s(v')}{\beta} \leq \frac{4\sqrt{d}s(v')}{\beta}$. Therefore, we will have $size(B(u')) \geq c_0 r(u', v)$ if we choose $c_0 \leq \frac{1}{2\sqrt{d}}$.

(b) $E'(v')$ does not intersect $B(u')$. In this case, we have $r(u', v') \geq \frac{2s(v')}{\beta}$ (by the fact that $E'(v')$ is centered at $l(v')$ and with a size $\frac{4s(v')}{\beta}$). Since v' is not removed from L in Step 2 when processing u', the diameter $D(u')$ of $B(u')$ must exceed $r(u', v')\frac{\beta}{2}$. Note that $s(v') \leq \frac{2s(v')}{\beta}$, and thus $s(v') \leq r(u', v')$. Then $r(u', v) \leq \|l(v) - l(v')\| + r(u', v') \leq 2r(u', v')$. This means that the diameter $D(u')$ of $B(u')$ exceeds $r(u', v')\frac{\beta}{2} \geq r(u', v)\frac{\beta}{4}$. From this, we immediately know $size(B(u')) \geq c_0 r(u', v)$ if $c_0 \leq \frac{\beta}{4\sqrt{d}}$.

2. u is generated in Step 5 in **Algorithm 1** when processing u'. In this case, $\Phi(u) = B(u)$ or $\Phi(u) = B_m(u)$. Let D denotes the diameter of $\Phi(u)$. Note $D \geq D(u')/4$ by definition of $B_m(\cdot)$ and **Lemma 7**. Let v' be the distance-node in L when processing u' which is either an ancestor of v or v itself. For this case, we also consider two sub-cases, depending on whether $E'(v')$ intersects $B(u')$.

(a) $E'(v')$ intersects $B(u')$. In this case, by exactly the same argument given above for Case 1(a), we know that the diameter $D(u')$ of $B(u')$ is at least $\frac{r(u', v)}{2}$. Then, $r(u, v) \leq D(u') + r(u', v) \leq 3D(u')$. Also note that $D(u') \leq 4D(u)$. Thus, $D(u) \geq \frac{r(u,v)}{12}$. In this case, we can choose $c_0 \leq \frac{1}{12\sqrt{d}}$.

(b) $E'(v')$ does not intersect $B(u')$. By the same argument given above for Case 1(b), we know $D(u') \geq r(u', v)\frac{\beta}{4}$. Thus, $r(u, v) \leq D(u') + r(u', v) \leq \frac{4+\beta}{\beta}D(u')$. Since $D(u') \leq 4D(u)$, we have $D(u) \geq \frac{\beta}{16+4\beta}$. This means that we can choose $c_0 \leq \frac{\beta}{(16+4\beta)\sqrt{d}}$.

By the above discussion, we know that if we choose c_0 as the minimum of the 4 possible choices, we have the desired bound $s \geq c_0 r$ for the size s of each box in $\mathcal{B}$. This means that the theorem then follows from **Lemma 9**. □

[1] S. Arya, T. Malamatos, and , D. M. Mount, Space-time tradeoffs for approximate nearest neighbor searching, *Journal of the ACM (JACM)*, 57(1)(2009), 1-54.

[2] G. Barequet, M.T. Dickerson, and R.L.S. Drysdale III, 2-Point Site Voronoi Diagrams, *Discrete Applied Mathematics*, **122(1-3)**(2002) 37-54.

[3] P. Callahan and R. Kosaraju, A Decomposition of Multidimensional Point Sets with Applications to k-nearest-neighbors and n-body Potential Fields, *JACM*, 42(1)(1995), 67-90.

[4] P. Cheilaris, E. Khramtcova, S. Langerman, and E. Papadopoulou, A Randomized Incremental Approach for the Hausdorff Voronoi Diagram of Non-crossing Clusters. *LATIN 2014* (2014) pp. 96-107.

[5] D. Z. Chen, Z. Huang, Y. Liu and J. Xu, On Clustering Induced Voronoi Diagrams, in *Proc. 54th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS 2013)* (2013) pp. 390-399.

[6] D. Z. Chen, Z. Huang, Y. Liu and J. Xu, On Clustering Induced Voronoi Diagrams, *SIAM Journal on Computing*, 46(6): 1679-1711, 2017.

[7] S. Har-Peled, A replacement for Voronoi diagrams of near linear size, in *Proceedings of FOCS 2001* (2001), pp. 94103.

[8] S. Har-Peled, Geometric approximation algorithms. Vol. 173. Boston: American mathematical society (2011).

[9] D. Hodorkovsky, 2-Point Site Voronoi Diagrams, M.Sc. Thesis, Technion, Haifa, Israel, (2005).

[10] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, 2nd Eds., (John Wiley & Sons, 2000).

[11] E. Papadopoulou, The Hausdorff Voronoi Diagram of Point Clusters in the Plane, *Algorithmica*, **40** (2004), 63-82.

[12] V. Polianskii and F.T. Pokorny, Voronoi boundary classification: A high-dimensional geometric approach via weighted monte carlo integration. In *International Conference on Machine Learning (2019)* pp. 5162–5170.

[13] J. Wang, J.D. MacKenzie, R. Ramachandran, and D.Z. Chen, Identifying Neutrophils in H&E Staining Histology Tissue Images, in *Proc. of the 17th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI), Part I* (2014) pp. 73–80.

[14] J. Wang, J.D. MacKenzie, R. Ramachandran, and D.Z. Chen, Neutrophils Identification by Deep Learning and Voronoi Diagram of Clusters, in *Proc. 18th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI), Part III* (2015) pp. 226-233.

[15] J. Xu, L. Xu, and E. Papadopoulou, Computing the Map of Geometric Minimal Cuts. *Algorithmica*, **68(4)** (2014) 805-834.